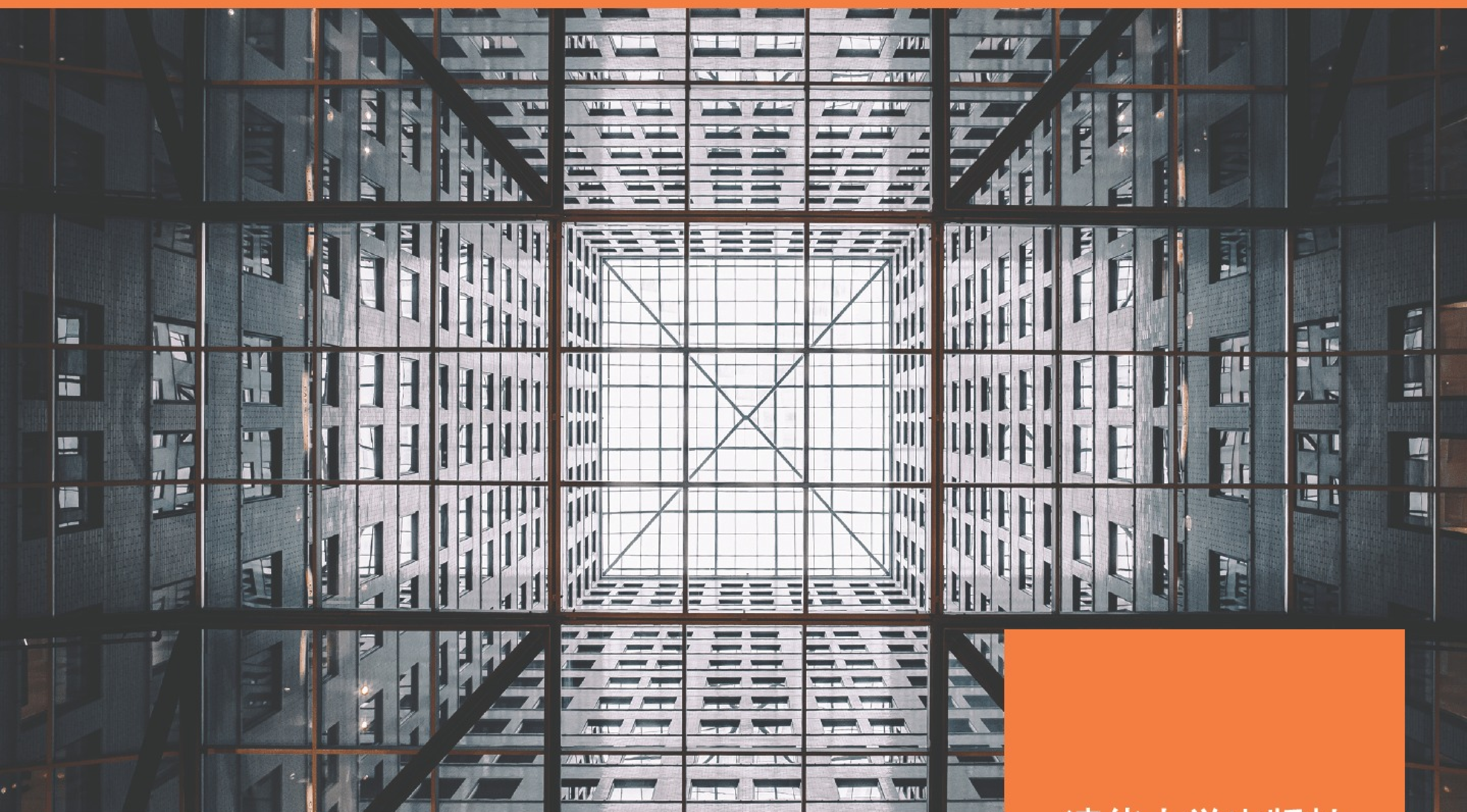
Mastering Spring Cloud

# 精通Spring Cloud微服务架构

[美] 皮奥特·闵可夫斯基 著　　黄进青 译



清华大学出版社

# 精通 Spring Cloud 微服务架构

[美] 皮奥特·闵可夫斯基　著

黄进青　译

清华大学出版社

北　京

## 内 容 简 介

本书详细阐述了与 Spring Cloud 微服务框架相关的基本解决方案，主要包括微服务简介、使用微服务的 Spring、Spring Cloud 概述、服务发现、使用 Spring Cloud Config 进行分布式配置、微服务之间的通信、高级负载均衡和断路器、使用 API 网关进行路由和过滤、分布式日志记录和跟踪、其他配置和发现功能、消息驱动的微服务、保护 API 的安全、测试 Java 微服务、Docker 支持、云平台上的 Spring 微服务等内容。此外，本书还提供了相应的示例、代码，以帮助读者进一步理解相关方案的实现过程。

本书适合作为高等院校计算机及相关专业的教材和教学参考书，也可作为相关开发人员的自学教材和参考手册。



**图书在版编目（CIP）数据**

精通 Spring Cloud 微服务架构/（美）皮奥特·闵可夫斯基（Piotr Minkowski）著；黄进青译. —北京：清华大学出版社，2019

书名原文：Mastering Spring Cloud

ISBN 978-7-302-53025-1

Ⅰ. ①精…  Ⅱ. ①皮…  ②黄…  Ⅲ. ①互联网络-网络服务器  Ⅳ. ①TP368.5

中国版本图书馆 CIP 数据核字（2019）第 093915 号

**责任编辑：**贾小红
**封面设计：**刘 超
**版式设计：**魏 远
**责任校对：**马子杰
**责任印制：**沈 露

**出版发行：**清华大学出版社
**网 址：**http://www.tup.com.cn，http://www.wqbook.com
**地 址：**北京清华大学学研大厦 A 座  **邮 编：**100084
**社 总 机：**010-62770175  **邮 购：**010-62786544
**投稿与读者服务：**010-62776969，c-service@tup.tsinghua.edu.cn
**质 量 反 馈：**010-62772015，zhiliang@tup.tsinghua.edu.cn
**印 装 者：**三河市铭诚印务有限公司
**经 销：**全国新华书店
**开 本：**185mm×230mm  **印 张：**23.75  **字 数：**474 千字
**版 次：**2019 年 7 月第 1 版  **印 次：**2019 年 7 月第 1 次印刷
**定 价：**119.00 元

产品编号：080090-01

# 译 者 序

在过去的 2018 年，有两个词汇经常挂在开发人员的嘴上，一个是“微信小程序”，一个是“微服务架构”，而能把它们连接在一起的，有一个更热门的词汇，那就是 Spring Cloud。

“微服务”的概念是由 ThoughtWorks 公司的首席科学家 Martin 提出的，他是敏捷开发方法的创始人之一，自然而然地，微服务的目的就是有效拆分应用，实现敏捷开发和部署。与之相对应的，此前的软件开发都是基于一体化架构。一体化架构的优点是开发简单直接、集中式管理、功能都在本地；但是，一旦应用程序变大、团队扩张，那么一体化架构的缺点就会立即凸显，庞大的一体代码库可能会让新手程序员望而生畏，再加上开发效率低、代码维护困难、部署不灵活、难以扩展应用等，这些很可能成为企业和开发人员难以逾越的障碍。微服务的出现解决了这些矛盾，它将系统拆分成进程独立的服务，进行分布式管理、自动化运维，通过 API 网关、服务间调用、服务发现、服务容错、服务部署和数据调用实现了开发简单、独立按需扩展、高可用性、持续集成和持续交付等，特别契合云平台应用程序开发的需要，这也正是它日益流行的原因。

Spring Cloud 是微服务架构开发的完美解决方案，它是一套分布式服务治理的框架，专注于全局微服务协调整理，可以将各个单独的微服务整合并管理起来，为各个微服务之间提供配置管理、服务发现、断路器、路由、消息代理、事件总线、决策竞选、分布式会话等集成服务。Spring Cloud 本身不提供具体功能性的操作，更专注于服务之间的通信、熔断和监控等，因此就需要很多组件来支持完整功能。本书介绍的 Spring Cloud 的可用组件及其主要功能包括：Spring Cloud Netflix Eureka（服务发现）、Spring Cloud Config（分布式配置）、Spring RestTemplate 和 Feign 客户端（服务间通信）、Ribbon（负载均衡算法）、Hystrix（断路器模式和仪表板监控）、Spring Cloud Netlix Zuul（路由和过滤）、Spring Cloud Sleuth（分布式服务跟踪）、Consul 和 ZooKeeper（服务发现和分布式配置）、RabbitMQ 和 Apache Kafka（消息代理）、Spring Cloud Contract（契约测试）、Gatling（自动化测试）、Jenkins（持续集成服务器）和 Kubernetes 平台等。此外，本书还介绍了 Docker 容器和两个支持 Java 应用程序的流行云平台：Pivotal Cloud Foundry 和 Heroku。相信在阅读本书之后，读者会对 Spring Cloud 和 Spring Boot 框架的应用和开发有一个高屋建瓴的认识，并掌握各个组件的应用技巧，熟练驾驭微服务应用程序的开发。

在翻译本书的过程中，为了更好地帮助读者理解和学习，本书以中英文对照的形式保留了大量的术语，这样的安排不但方便读者理解书中的代码，而且也有助于读者查找和利用本书配套网站上的资源。

本书由黄进青翻译，马宏华、唐盛、郝艳杰、黄永强、陈凯、熊爱华、黄刚等也参与了部分翻译工作。由于译者水平有限，错漏之处在所难免，在此诚挚欢迎读者提出任何意见和建议。

译　者

# 前　　言

开发、部署和运营云应用程序应该像本地应用程序一样简单。这应该是任何云平台、库或工具背后的管理原则。Spring Cloud 可以轻松地为云开发 JVM 应用程序。本书将介绍 Spring Cloud 并帮助开发人员掌握其功能。

本书首先介绍如何配置 Spring Cloud 服务器并运行 Eureka 服务器以启用服务注册和发现；然后再深入剖析与负载均衡和断路相关的技术，包括利用 Feign 客户端的所有功能；最后讨论和研究高级主题，包括如何为 Spring Cloud 实现分布式跟踪解决方案并构建消息驱动的微服务架构。

## 本书适合的读者

本书对热衷于利用 Spring Cloud 的开发人员有很强的吸引力。Spring Cloud 是一个开源库，可帮助开发人员快速构建分布式系统。了解 Java 和 Spring Framework 将对本书的学习很有帮助，但之前不需要接触 Spring Cloud。

## 本书内容综述

本书的写作思路明确，结构简单易懂。全书共分为 3 个部分，第一部分是“微服务架构和 Spring Cloud 项目基础知识”，包括第 1 章～第 3 章，详细介绍了微服务、Spring Boot 和 Spring Cloud 的基础知识。

- 第 1 章“微服务简介”，将介绍微服务架构、云环境等。读者将学习并理解基于微服务的应用程序和一体化应用程序之间的区别，同时了解如何迁移到微服务应用程序。
- 第 2 章“使用微服务的 Spring”，将介绍 Spring Boot 框架。本章将详细说明如何有效地使用 Spring Boot 框架来创建微服务应用程序。此外还将介绍使用 Spring MVC 注解创建 REST API、使用 Swagger2 提供 API 文档，以及使用 Spring Boot

Actuator 端点公开运行状况检查和指标数据等主题。

- 第 3 章“Spring Cloud 概述”，将简要介绍作为 Spring Cloud 一部分的主要项目。它将侧重于说明 Spring Cloud 实现的主要模式并将它们分配给特定项目。

本书的第二部分是“微服务架构常见元素和 Spring Cloud 实现”，包括第 4 章～第 13 章，详细介绍了 Spring Cloud 各个组件的配置和应用。

- 第 4 章“服务发现”，将使用 Spring Cloud Netflix Eureka 描述服务发现模式。本章将详细说明如何在独立模式下运行 Eureka 服务器，以及如何使用对等副本运行多个服务器实例。此外还将介绍如何在客户端启用发现并在不同区域中注册这些客户端。
- 第 5 章“使用 Spring Cloud Config 进行分布式配置”，将详细介绍如何在应用程序中使用 Spring Cloud Config 进行分布式配置。本章将说明如何使用 Spring Cloud Bus 启用属性源的不同后端存储库并推送更改通知。通过比较发现第一个引导程序和配置第一个引导程序方法，详细说明了发现服务和配置服务器之间的集成。
- 第 6 章“微服务之间的通信”，将描述参与服务间通信的最重要元素：HTTP 客户端和负载均衡器。本章将详细介绍如何在有或没有服务发现的情况下使用 Spring RestTemplate、Ribbon 和 Feign 客户端。
- 第 7 章“高级负载均衡和断路器”，将描述与微服务之间的服务间通信相关的更高级主题。本章将详细介绍如何使用 Ribbon 客户端实现不同的负载均衡算法，使用 Hystrix 启用断路器模式并使用 Hystrix 仪表板监控通信统计信息。
- 第 8 章“使用 API 网关进行路由和过滤”，将比较用作 Spring 云应用程序的 API 网关和代理的两个项目：Spring Cloud Netlix Zuul 和 Spring Cloud Gateway。本章将详细介绍如何将它们与服务发现集成，并创建简单而更高级的路由和过滤规则。
- 第 9 章“分布式日志记录和跟踪”，将介绍一些流行的工具，用于收集和分析由微服务生成的日志记录和跟踪信息。本章将说明如何使用 Spring Cloud Sleuth 附加跟踪信息和关联消息，此外还将运行与 Elastic Stack 集成的示例应用程序，以便发送日志消息，并使用 Zipkin 来收集跟踪的信息。
- 第 10 章“其他配置和发现功能”，将介绍两种用于服务发现和分布式配置的流行产品：Consul 和 ZooKeeper。本章将详细说明如何在本地运行这些工具，并将 Spring Cloud 应用程序与它们集成在一起。
- 第 11 章“消息驱动的微服务”，将指导开发人员如何在微服务之间提供异步的、消息驱动的通信。本章将详细介绍如何将 RabbitMQ 和 Apache Kafka 消息代理与

Spring Cloud 应用程序集成，以实现异步一对一和发布/订阅通信方式。

- 第 12 章“保护 API 的安全”，将描述保护微服务的各种方法。本章将实现一个由所有先前引入的元素组成的系统，这些元素通过 SSL 相互通信。此外还将详细说明如何使用 OAuth2 和 JWT 令牌来给传入 API 的请求授权。
- 第 13 章“测试 Java 微服务”，将描述微服务测试的不同策略。它将侧重于演示由使用者驱动的契约测试，这尤其适用于基于微服务的环境。此外还将介绍如何使用 Hoverfly、Pact、Spring Cloud Contract、Gatling 等框架来实现不同类型的自动化测试。

本书的第三部分是“Docker 支持和 Spring Cloud 平台”，包括第 14 章～第 15 章，详细介绍了 Docker 容器、Pivotal Cloud Foundry 和 Heroku 云平台。

- 第 14 章“Docker 支持”，将简要介绍 Docker。它将侧重于描述最常用的 Docker 命令，这些命令用于在容器化环境中运行和监视微服务。此外还将详细说明如何使用流行的持续集成服务器（Jenkins）构建和运行容器，并将它们部署在 Kubernetes 平台上。
- 第 15 章“云平台上的 Spring 微服务”，将介绍两个支持 Java 应用程序的流行云平台：Pivotal Cloud Foundry 和 Heroku。本章将详细说明如何使用命令行工具或 Web 控制台在这些平台上部署、启动、扩展和监视应用程序。

## 阅读基础

要顺利阅读本书并完成所有代码示例，读者应具备以下基础条件：

- 有效的互联网连接
- Java 8+
- Docker
- Maven
- Git 客户端

## 下载示例代码文件

读者可以从 www.packtpub.com 下载本书的示例代码文件。具体步骤如下：

（1）登录或注册 www.packtpub.com。

（2）选择 **Support**（支持）选项卡。

（3）单击 **Code Downloads＆Errata**（代码下载和勘误表）。

（4）在 **Search**（搜索）框中输入图书名称 Mastering Spring Cloud，然后按照屏幕上的说明进行操作。

下载文件后，请确保使用最新版本解压缩或解压缩文件夹：

- WinRAR/7-Zip（Windows 系统）
- Zipeg/iZip/UnRarX（Mac 系统）
- 7-Zip/PeaZip（Linux 系统）

该书的代码包也已经在 GitHub 上托管，网址为 https://github.com/PacktPublishing/Mastering-Spring-Cloud，欢迎访问。

## 本书约定

本书中使用了许多文本约定。

（1）CodeInText：表示文本中的代码字、数据库表名、文件夹名、文件名、文件扩展名、路径名、虚拟 URL、用户输入和 Twitter 句柄等。以下段落就是一个示例。

“HTTP API 端点（http://localhost:8889/client-service-zone3.yml）的最后一个可用版本，返回与输入文件相同的数据。”

（2）有关代码块的设置如下所示。

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-config-server</artifactId>
</dependency>
```

（3）当我们希望引起读者对代码块的特定部分的注意时，相关的行或项目以粗体显示。

```
spring:
 rabbitmq:
  host: 192.168.99.100
  port: 5672
```

（4）任何命令行输入或输出都采用如下所示的粗体代码形式。

```
$ curl -H “X-Vault-Token: client” -X GET
http://192.168.99.100:8200/v1/secret/client-service
```

（5）本书还使用了以下两个图标。

表示警告或重要的注意事项。

表示提示或小技巧。

## 关于作者

Piotr Mińkowski 拥有超过 10 年的银行和电信行业开发人员和架构师的工作经验。他擅长 Java 以及与之相关的技术、工具和框架。目前他在波兰的移动运营商 Play 公司工作，负责 IT 系统架构。他帮助该公司完成了从一体化应用程序/面向服务的架构（SOA）到基于微服务的架构的迁移工作，还帮助建立了完整的持续集成和持续交付环境。

## 关于审稿者

Samer ABDELKAFI 拥有超过 13 年的软件架构师和工程师工作经验，主要专注于开源技术。他为不同领域的众多项目做出了贡献，如银行、保险、教育、公共服务和公用事业计费等。2016 年年底，他创建了 DEVACT，一家专门从事信息技术咨询的公司。除了日常工作之外，Samer 还经常在他的博客中分享自己的经验，撰写与 Java 和 Web 技术相关的文章。

# 目　　录

## 第一部分　微服务架构和 Spring Cloud 项目基础知识

## 第二部分　微服务架构常见元素和 Spring Cloud 实现

## 第三部分 Docker 支持和 Spring Cloud 平台

# 第一部分

## 微服务架构和 Spring Cloud 项目基础知识

# 第 1 章　微服务简介

微服务是过去几年中 IT 界出现的最热门趋势之一。它们为什么会日益受到欢迎呢？要理解这一点其实相当简单，因为它们的优点和缺点都是众所周知的，而所谓的缺点却可以使用正确的工具轻松解决。它们所具有的优势包括可扩展性、灵活性和独立交付，这些都是其迅速普及的原因。另外，还有一些早期的 IT 趋势也对微服务的普及有所影响，这里所说的趋势是指常见的基于云的环境的使用，以及从关系数据库到 NoSQL 的迁移等。

本章将要讨论的主题包括：

- 使用 Spring Cloud 进行云原生开发。
- 基于微服务架构中最重要的元素。
- 服务间通信模型。
- 断路器和后备模式介绍。

## 1.1　微服务的优点

微服务（Microservices）的概念定义了 IT 系统体系结构的方法，该方法将应用程序划分为实现业务需求的松散耦合服务的集合。实际上，这是面向服务的架构（Service-Oriented Architecture，SOA）概念的变体。迁移到基于微服务的体系结构的最重要的好处之一是能够连续交付大型复杂应用程序。

截至目前，开发人员可能已经有很多机会阅读到有关微服务的书籍或文章，大多数资料都会详细描述它们的优点和缺点。在笔者看来，使用微服务确实有许多优点。首先，对于项目中的新开发人员来说，微服务相对较小且易于理解。开发人员通常希望确保在一个地方执行的代码修改不会对应用程序的所有其他模块产生不良影响。采用微服务之后，开发人员对这一点可谓深具信心，因为微服务可以做到仅实现一个业务领域，这和一体化（Monolithic）应用程序是不一样的，一体化应用程序有时甚至需要将各种看似无关的功能和插件等都整合到同一个软件包里。当然，这不是全部，笔者还注意到，一般来说，在小型微服务中维护高质量代码比在一体化应用程序中要容易得多，因为一体化应用程序通常会有许多开发人员介入修改。

关于微服务架构，值得嘉许的第二件事是其业务划分。到目前为止，当开发人员不得不处理复杂的企业系统时，总是会将系统划分为子系统，而这个划分是根据其他子系统完成的。例如，电信企业总是有一个计费子系统，开发人员会创建一个隐藏计费复杂性的子系统并对外提供应用程序编程接口（Application Programming Interface，API），然后开发人员会发现自己需要数据，但这些数据无法存储在计费系统中，因为它不容易自定义，所以开发人员需要再创建另一个子系统，这将导致开发人员只能构建一个复杂的子系统网格，这个子系统网格很不容易理解，对于企业中的新员工来说尤其如此。使用微服务则不会遇到这样的问题，如果它们设计得很好，那么每个微服务都应该负责整个选定的业务领域。在某些情况下，无论企业活跃的部门如何，这些领域都是相似的。

## 1.2　使用 Spring Framework 构建微服务

尽管微服务的概念多年来一直是一个重要的主题，但仍然没有很多稳定的框架支持运行完整微服务环境所需的所有功能。自从笔者开始使用微服务之后，就一直在努力跟上最新的框架并找出针对微服务需求而开发的功能。当然还有一些其他有趣的解决方案，如 Vert.x 或 Apache Camel，但它们都不是 Spring Framework 的良配。

Spring Cloud 实现了基于微服务架构中使用的所有经过验证的模式，如服务注册表（Service Registries）、配置服务器（Configuration Server）、断路器（Circuit Breakers）、云总线（Cloud Buses）、OAuth2 模式（OAuth2 Patterns）和 API 网关（API Gateways）。它拥有强大的网络讨论社区，因此其新功能会以高频率发布。它基于 Spring 的开放式编程模型，该模型被全球数百万 Java 开发人员使用。它也有编写得非常优秀的说明文档。在线查找许多可用的 Spring Framework 使用示例时，开发人员不会遇到任何问题。

## 1.3　云原生应用程序开发方法

微服务本质上与云计算平台相关联，但微服务的实际概念并不是什么新鲜事。这种方法已经在 IT 开发领域应用了很多年，现在，通过云解决方案的普及，它已经发展到更高的水平。所以，不难理解这种受欢迎程度的原因。与企业的内部部署（On-Premise）解决方案相比，云的使用为开发人员提供了可扩展性、可靠性和低维护成本等优点，这导致云原生应用程序开发方法（Cloud-native Application Development Approach）的兴起，旨在为开发人员提供类似云的弹性扩展、不可变部署和一次性实例具有的所有优势。这

一切都归结为一件事——减少满足新要求所需的时间和成本。如今，软件系统和应用程序正在不断改进。如果开发人员采用传统的开发方法，即基于一体化开发方式，则代码库会持续增长并且变得过于复杂，以至于无法进行修改和维护，引入新功能、框架和技术也变得很难，这反过来会影响创新并抑制新想法。在这一点上，开发人员无疑深有同感。

这枚硬币还有另一面。今天几乎每个人都在考虑迁移到云端，部分原因是它显得很时尚。每个人都需要这个吗？当然不是。那些不确定是否要将应用程序迁移到远程云提供商（如 AWS、Azure 或 Google）的人，至少希望拥有内部部署的私有云或 Docker 容器（Docker Container）。但是，无论是迁移到远程云端还是部署私有云服务器，都需要不小的成本开支，它真的能带来足够的好处吗？在讨论云原生开发和云平台之前，值得思考这个问题。

我并不是想阻止开发人员使用 Spring Cloud，恰恰相反，我们必须彻底了解云原生的开发才能做到物超所值。以下是对于云原生应用程序开发的一个非常好的定义：

“原生云应用程序是专为云计算环境设计的程序，而不是简单地迁移到云计算。”

Spring 旨在加速开发人员的云原生开发。使用 Spring Boot 构建应用程序非常快。本书第 2 章将详细介绍如何做到这一点。Spring Cloud 实现了微服务架构模式，并且可以帮助开发人员使用该领域最流行的解决方案。使用这些框架开发的应用程序可以轻松地调整为部署在 Pivotal Cloud Foundry（PCF）或 Docker 容器上，但它们也可以按传统方式在一台或多台计算机上作为独立进程启动，并且开发人员将拥有微服务方法的优势。接下来不妨深入探讨一下微服务架构。

## 1.4　了解微服务架构

现在假设有一个客户要求设计一个解决方案。他们需要某种银行应用程序来保证整个系统内的数据一致性。到目前为止，该客户一直在使用 Oracle 数据库，并且还购买了技术支持。在不考虑太多的情况下，我们可以选择基于关系数据模型设计一个一体化应用程序。现在先来看一个系统设计的简化示意图，如图 1.1 所示。

有 4 个实体映射到数据库中的表：

- 第一个是客户（Customer）实体，它将存储和检索活动客户端列表。
- 每个客户可以有一个或多个账户，由账户（Account）实体操作。

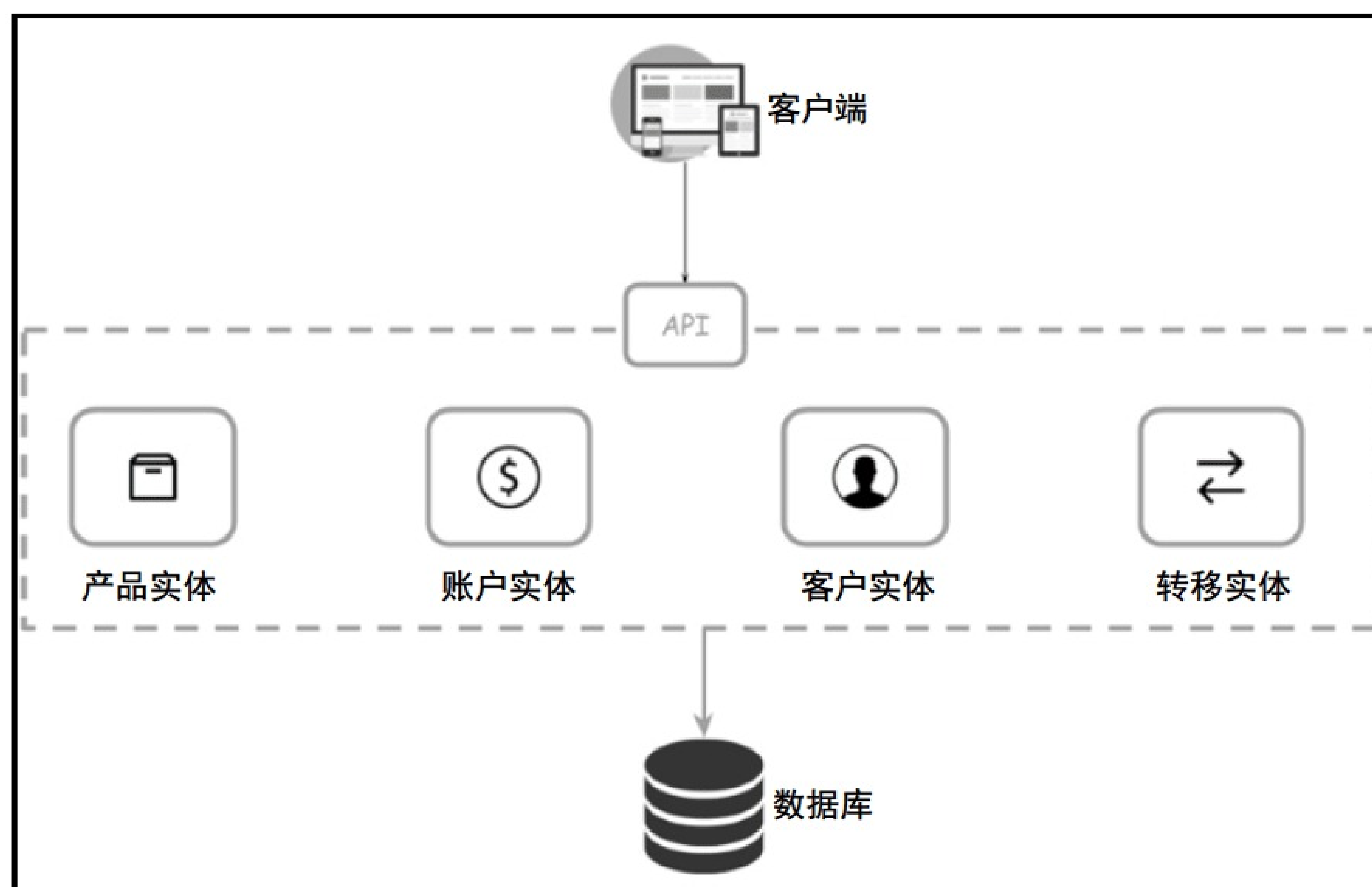



图 1.1　系统设计简化示意图

- 转移（Transfer）实体负责在系统内的账户之间执行所有资金转移汇兑。
- 还有一个产品（Product）实体，创建该实体可用于存储客户的存款和分配给客户的信用额度等信息。

在没有进一步详细说明的情况下，该应用程序公开了 API，该 API 提供了在设计的数据库上实现功能所需的所有操作。当然，该实现将符合三层模型。

一致性不再是最重要的要求，它甚至不是强制性的。客户需要一个解决方案，但并不希望这个解决方案的开发强行要求重新部署整个应用程序。它应该是可扩展的，并且应该能够轻松扩展新的模块和功能。此外，客户也不会向开发人员施加压力，要求他们必须使用 Oracle 或其他关系数据库——不仅如此，他还乐于避免使用它。这些理由足以决定迁移到微服务吗？我们假设已经足够。接下来，开发人员可以将一体化应用程序划分为 4 个独立的微服务，每个微服务都有一个专用的数据库。在某些情况下，它仍然可以是关系数据库，而在其他情况下，它也可以是 NoSQL 数据库。现在，我们的系统包含许多独立构建的服务，并将在我们的环境中运行。随着微服务数量的增加，系统复杂性也在不断提高。我们希望隐藏外部 API 客户端的复杂性，外部 API 客户端不应该知道它与服务 X 而不是服务 Y 进行通信。网关负责将所有请求动态路由到不同的端点。在这里，动态（Dynamically）一词意味着它应该基于服务发现（Service Discovery）中的条目，本

章后面的第 1.4.1 节“理解服务发现的必要性”将对此有详细的讨论。

隐藏特定服务或动态路由的调用并不是 API 网关的唯一功能。由于它是我们系统的入口点，因此它可以是跟踪重要数据、收集请求的指标性数据以及计算其他统计数据的好地方。它可以丰富请求头（Request Header）或响应头（Response Header），以便包含可供系统内应用程序使用的其他一些信息。它应该执行一些安全操作，如身份验证和授权，并且应该能够检测每个资源的需求并拒绝不满足它们的请求。如图 1.2 所示就是这样一个系统示例，它由 4 个独立的微服务组成，它们对于在 API 网关后面的外部客户端来说是隐藏的。

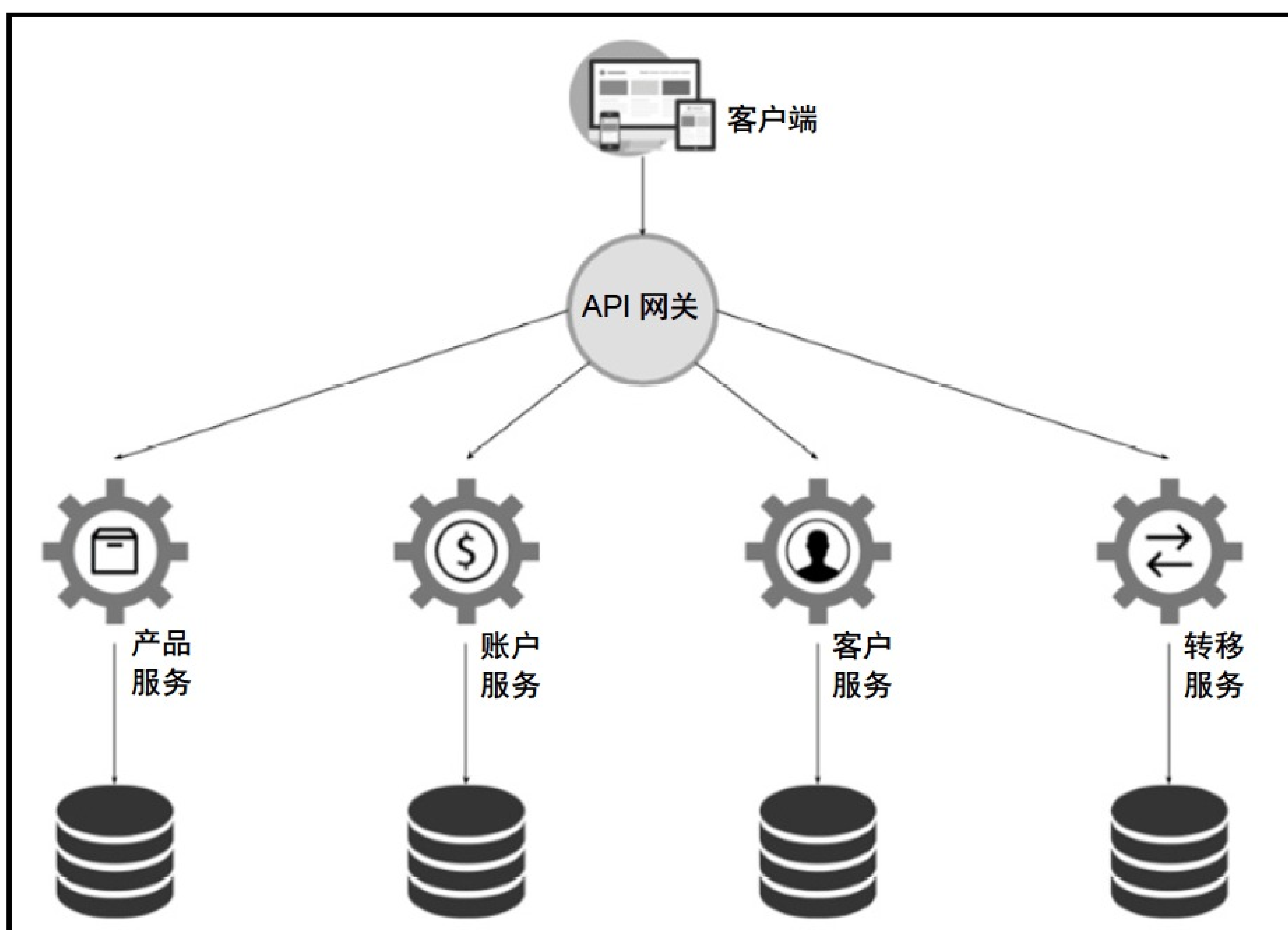



图 1.2　微服务和 API 网关组成示意图

## 1.4.1　理解服务发现的必要性

现在假设开发人员已经将一体化应用程序划分为更小的独立服务。从外部看，我们的系统看起来仍然和以前一样，因为它的复杂性隐藏在 API 网关之后。实际上，目前的微服务并不算多，但以后也可能会有更多。此外，每个微服务都可以与其他微服务交互。

这意味着每个微服务都必须保留有关其他微服务的网络地址的信息。保持这样的配置可能非常麻烦，特别是当涉及手动覆盖每个配置时。如果这些地址在重启后动态变化了该怎么办？图 1.3 显示了我们的示例微服务之间的调用路由。

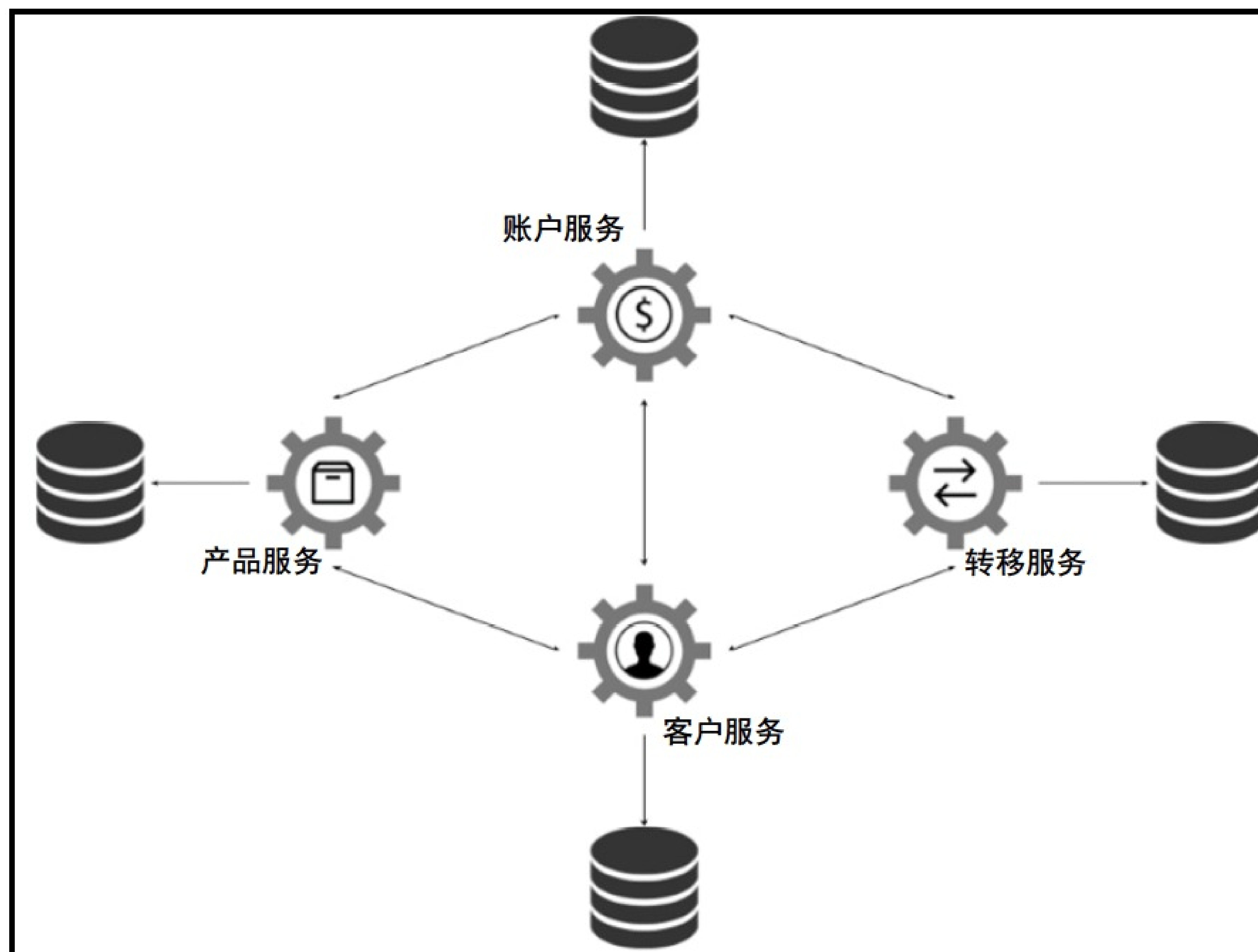



图 1.3　示例微服务之间的调用路由示意图

服务发现（Service Discovery）可以自动检测这些设备在计算机网络上提供的设备和服务。在基于微服务架构的情况下，这是必要的机制。启动后的每个服务都应该在一个中心位置注册，这个中心位置可以由所有其他服务访问。注册键（Registration Key）应该是服务或标识符的名称，在整个系统中必须是唯一的，以便其他微服务能够使用该名称查找和调用该服务。具有给定名称的每个键都分配了一些值。在最常见的情况下，这些属性（Attribute）将指示服务的网络位置。更准确地说，它们表示微服务的一个实例，因为它可以作为在不同机器或端口上运行的独立应用程序而倍增。有时也可以发送一些其他信息，但这取决于具体的服务发现提供程序。但是，这里重要的是在一个键下，可以注册同一个服务的多个实例。除注册外，每个服务还将获取在特定发现服务器上注册的其他服务的完整列表。不仅如此，每个微服务都必须了解注册列表中的任何更改。这可

以通过定期更新之前从远程服务器收集的配置来实现。

某些解决方案会将服务发现的使用与服务器配置功能相结合。当开发人员真正面对它时就会明白，这两种方法非常相似。通过服务器配置，开发人员可以集中管理系统中的所有配置文件。一般来说，这样的配置就是服务器作为表述性状态传递（Representational State Transfer，REST）Web 服务。在启动之前，每个微服务都会尝试连接到服务器并获取专门为其准备的参数。其中一种方法是将这种配置存储在版本控制系统中——如 Git（Git 是目前最为先进的分布式版本控制系统），然后配置服务器更新其 Git 工作副本并将所有属性作为 JSON 提供。在另一种方法中，我们可以使用存储键值对（Key-Value Pairs）的解决方案，并在服务发现过程中履行提供者的角色。最受欢迎的工具是 Consul 和 Zookeeper。图 1.4 给出了一个系统的架构，该系统由一些带有数据库后端的微服务组成，这些服务在一个称为发现服务（Discovery Service）的中央服务中注册。

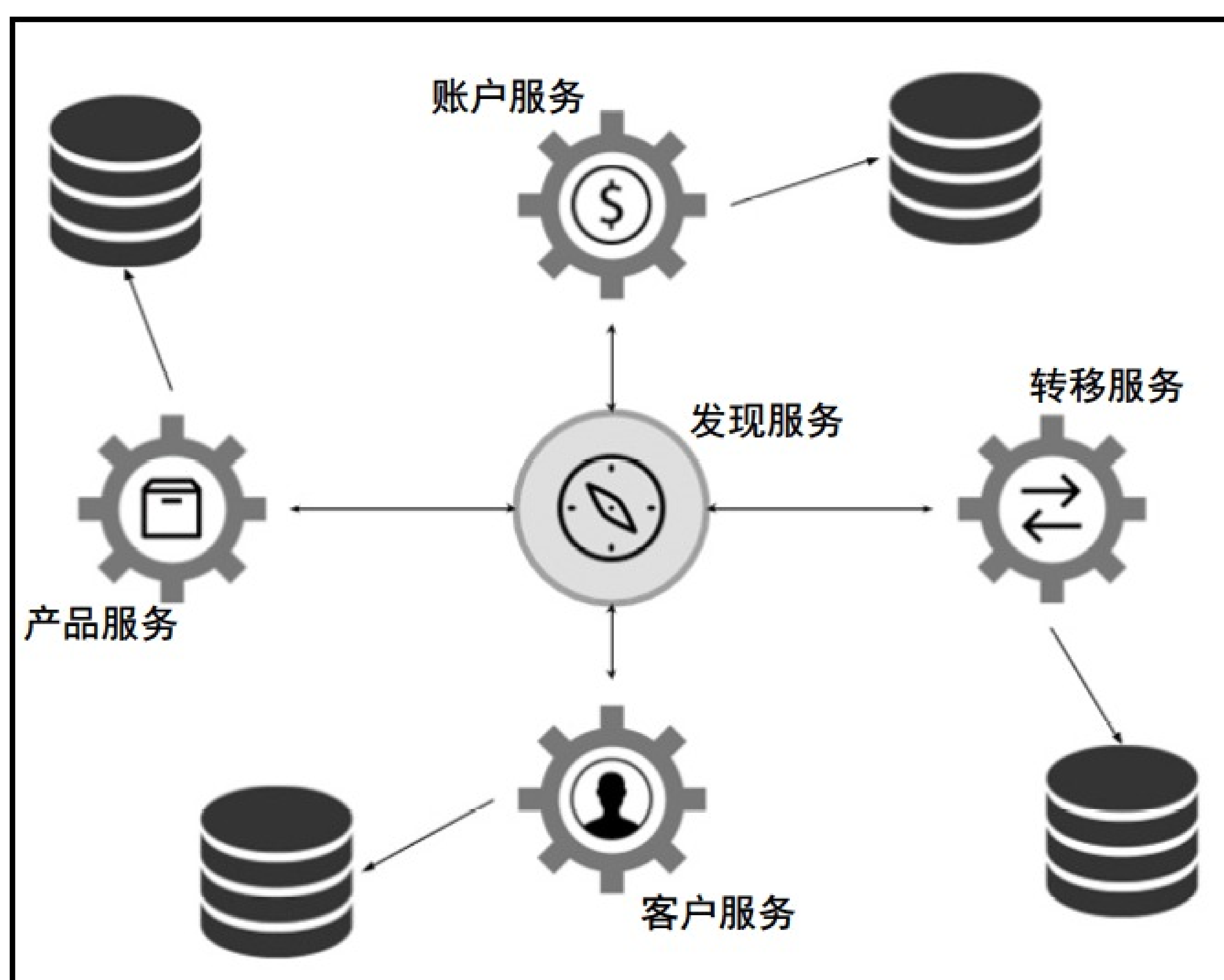



图 1.4　示例微服务均将在发现服务中注册

## 1.4.2　服务之间的通信

为了保证系统的可靠性，开发人员不能允许每个服务只有一个实例运行的情况。一般来说，每个服务至少需要有两个实例运行，以防其中一个实例出现故障。当然，也可能会有更多，但出于性能考虑，开发人员会保持较低水平。无论如何，同一服务的多个

实例必须对传入的请求使用负载均衡。首先，负载均衡器（Load Balancer）通常内置在 API 网关中。此负载均衡器应从发现服务器获取已注册实例的列表。如果没有理由不这样做，那么开发人员通常会使用轮询调度规则（Round-Robin Rule）来平衡所有正在运行的实例之间的传入流量为 50/50。同样的规则也适用于微服务端的负载均衡器。

图 1.5 说明了两个示例微服务的多个实例之间的服务间通信中所涉及的最重要的组件。

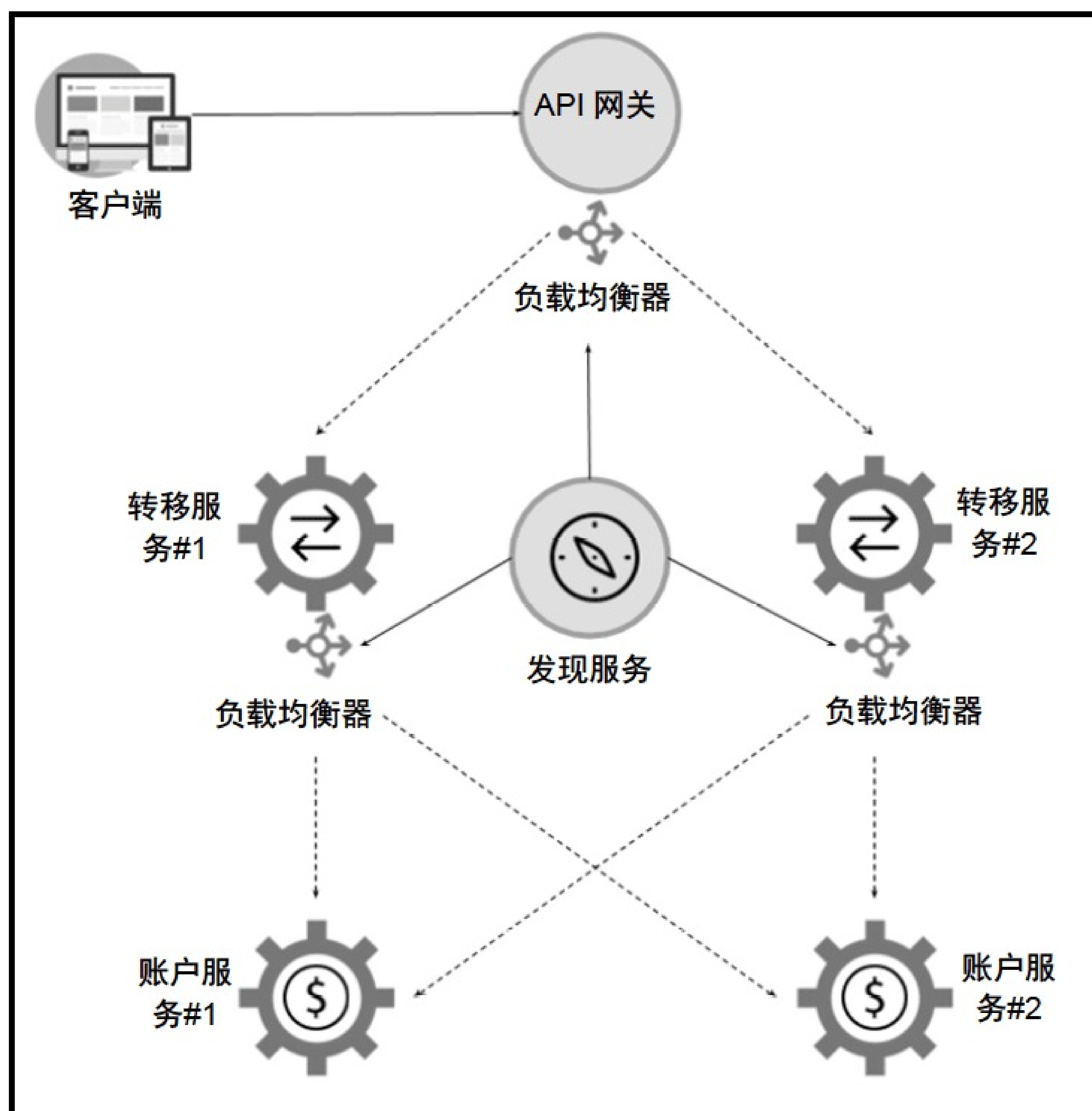



图 1.5　两个示例微服务的负载均衡器

大多数人在听到微服务时会认为它包含带有 JSON 表示法的 RESTful Web 服务，但这只是其中一种可能性。我们可以使用其他一些交互方式，当然，它们不仅适用于基于微服务的架构。应该执行的第一个分类是一对一或一对多通信。在一对一交互中，每个传入请求仅由一个服务实例处理，而在一对多交互中，它将由多个服务实例处理。但最流行的划分标准则为调用是同步的还是异步的。另外，异步通信还可以划分为通知信息。当客户端向服务发送请求但不期望回复时，它只要执行一个简单的异步调用即可，该异

步调用不会阻塞线程并以异步方式进行回复。

此外，值得一提的还有响应式微服务（Reactive Microservices）。从版本 5 开始，Spring 也已经支持这种类型的编程。还有一些库也支持响应式微服务，并且可以使用 NoSQL 数据库（如 MongoDB 或 Cassandra）进行交互。最后一个众所周知的通信类型是发布-订阅（Publish-Subscribe）。这是一对多交互类型，其中由客户端发布消息，然后由所有侦听服务使用。一般来说，此模型将使用消息代理（Message Broker）实现，常见的消息代理包括 Apache Kafka、RabbitMQ 和 ActiveMQ 等。

### 1.4.3　故障和断路器

前文已经讨论了与微服务架构相关的大多数重要概念。这些机制（包括服务发现、API 网关和配置服务器等）是帮助开发人员创建可靠且高效的系统的有用元素。当然，即使在设计系统架构时考虑过这些问题的许多方面，开发人员也应该始终为失败做好准备。在许多情况下，失败的原因完全超出了开发人员的控制范围，如网络或数据库问题。对于基于微服务的系统，这种错误可能特别严重，因为该系统中的一个输入请求可能需要在许多后续调用中被处理。应该优先考虑的良好做法是在等待响应时始终使用网络超时。如果单个服务存在性能问题，则应尽量减少对其余服务的影响。发送一个出错的响应消息比长时间等待回复，阻塞其他线程要好得多。

网络超时问题的一个有趣的解决方案可能是断路器模式（Circuit Breaker Pattern）。这是一个与微服务方法密切相关的概念。断路器负责计算成功和失败的请求。如果错误率超过假定阈值，则它会跳闸（这里采用了与电路相关的概念和术语）并导致所有进一步的尝试立即失败。在特定时间段之后，API 客户端应该返回发送请求，如果成功，则关闭断路器。如果每个服务有多个实例可用，并且其中一个服务的工作速度比其他服务慢，那么结果就是在负载平衡过程中忽略了它。处理部分网络故障的第二种常用机制是回退逻辑（Fallback Logic），这是在请求失败时必须执行的逻辑。例如，服务可以返回已缓存的数据、默认值或空结果列表。就个人而言，笔者不是这个解决方案的忠实粉丝，笔者宁愿将错误代码传递到其他系统，而不是返回已缓存数据或默认值。

## 1.5　小　　结

Spring Cloud 的一大优势是它支持本章所讨论过的所有模式和机制。与其他一些框架不同，它们也是稳定的实现。在本书第 3 章“Spring Cloud 概述”中将详细介绍 Spring Cloud

项目所支持的模式。

本章讨论了与微服务架构相关的最重要的概念，如云原生应用程序开发、服务发现、分布式配置、API 网关和断路器模式等。本章还以企业应用程序的开发为例，提出了对这种方法的优缺点的看法。然后，在第 1.4 节中描述了与微服务相关的主要模式和解决方案。其中一些是众所周知的模式，它们已经存在多年，但仍在某种程度上被视为 IT 世界的新事物。在本小结中，我们要强调的是，微服务本质上是云原生的。也就是说，它天然地适应云平台开发模式。Spring Boot 和 Spring Cloud 等框架都可以帮助开发人员加速云原生开发。

迁移到云原生开发的主要动机是能够在保持高质量的同时更快地实现和交付应用程序。在许多情况下，微服务都可以帮助开发人员实现这一目标，但有时一体化应用程序方法也并不是一个糟糕的选择。

虽然微服务是小而独立的单元，但它们是集中管理的。诸如网络位置、配置、日志文件和指标等信息都应该存储在一个中心位置。有各种类型的工具和解决方案可以提供所有这些功能。本书后面的几乎所有章节都将详细讨论它们。Spring Cloud 项目旨在帮助开发人员整合所有内容。衷心希望本书所提供的内容能够有效地帮助读者掌握 Spring Cloud。

# 第 2 章　使用微服务的 Spring

据笔者所知，从未接触过 Spring Framework 的 Java 开发人员可谓寥寥无几。实际上，Spring Framework 是由许多项目组成的，它可以与许多其他框架一起使用，开发人员迟早都将被迫尝试使用它。虽然 Spring Boot 的应用经验相当不常见，但它很快就获得了很高的人气。与 Spring Framework 相比，Spring Boot 是一个相对较新的解决方案。它的实际版本是 2，而不是 Spring Framework 的 5。那么，创建它的目的是什么？使用 Spring Boot 与使用标准 Spring Framework 方式运行应用程序究竟有什么区别呢？

本章将要讨论的主题包括：

- 使用启动器以启用项目的其他功能。
- 使用 Spring Web 库实现公开 REST API 方法的服务。
- 使用属性和 YAML 文件自定义服务配置。
- 详细说明并提供公开的 REST 端点的规范。
- 配置运行状况检查和监控功能。
- 使用 Spring Boot 配置文件以使应用程序适应不同模式运行。
- 使用 ORM 功能与嵌入式和远程 NoSQL 数据库进行交互。

## 2.1　关于 Spring Boot

Spring Boot 专门用于运行独立的 Spring 应用程序。它与简单的 Java 应用程序一样，使用 java -jar 命令。使 Spring Boot 与标准 Spring 配置不同的基本原因是简单性（Simplicity）。这种简单性与我们需要了解的第一个重要术语密切相关，即启动器（Starter）。启动器是一个可以包含在项目依赖项中的工件（Artifact）。它只是为必须包含在应用程序中的其他工件提供一组依赖项（Dependency），以实现所需的功能。以这种方式提供的包可以使用，这意味着开发人员不必配置任何东西即可使其工作。这带来了与 Spring Boot 相关的第二个重要术语：自动配置（Auto-Configuration）。启动器包含的所有工件都具有默认设置，可以使用属性或其他类型的启动器轻松覆盖。例如，如果开发人员在某个应用程序的依赖项中包含 spring-boot-starter-web，则它会嵌入一个默认 Web 容器，并在应用程序启动期间在默认端口上启动它。继续深入了解可知，Spring Boot 中的默认 Web 容器是 Tomcat，它从端口 8080 开始。开发人员可以通过声明应用程序属

性文件中的指定字段来轻松更改此端口，甚至还可以通过在项目依赖项中包含 spring-boot-starter-jetty 或 spring-boot-starter-undertow 来更改 Web 容器。

关于 Starter，也就是启动器，这里还是有必要再多介绍一些。它们的官方命名模式是 spring-boot-starter-*，其中，*就是指特定类型的启动器。Spring Boot 中有很多可用的启动器，本节将仅选取一些最受欢迎的启动器进行简要介绍（见表 2.1），这些启动器也将应用于本书后面章节所提供的示例中。

**表 2.1　常见启动器**

| 名　称 | 说　明 |
|---|---|
| spring-boot-starter | 核心启动程序，包括自动配置支持、日志记录和 YAML |
| spring-boot-starter-web | 允许开发人员构建 Web 应用程序，包括 RESTful 和 Spring MVC。使用 Tomcat 作为默认嵌入式容器 |
| spring-boot-starter-jetty | 在项目中包含 Jetty 并将其设置为默认的嵌入式 servlet 容器 |
| spring-boot-starter-undertow | 在项目中包含 Undertow，并将其设置为默认的嵌入式 servlet 容器 |
| spring-boot-starter-tomcat | 包含 Tomcat 作为嵌入式 servlet 容器。可以通过 spring-boot-starter-web 使用默认的 servlet 容器启动器 |
| spring-boot-starter-actuator | 包含项目中的 Spring Boot Actuator，它提供监视和管理应用程序的功能 |
| spring-boot-starter-jdbc | 包含带有 Tomcat 连接池的 Spring JBDC。特定数据库的驱动程序应由开发人员自己提供 |
| spring-boot-starter-data-jpa | 包含使用 JPA/Hibernate 与关系数据库交互所需的所有工件 |
| spring-boot-starter-data-mongodb | 包含与 MongoDB 交互以及在 localhost 上初始化 Mongo 客户端连接所需的所有工件 |
| spring-boot-starter-security | 包含项目中的 Spring Security，默认情况下为应用程序启用基本安全性 |
| spring-boot-starter-test | 允许使用 JUnit、Hamcrest 和 Mockito 等库创建单元测试 |
| spring-boot-starter-amqp | 包含项目的 Spring AMQP，并启动 RabbitMQ 作为默认的 AMQP 代理 |

如果开发人员对可用启动器的完整列表感兴趣，请参阅 Spring Boot 规范。现在，不妨回过头来讨论 Spring Boot 与 Spring Framework 标准配置之间的主要区别。正如前文所述，开发人员可以包含 spring-boot-starter-web，它会将 Web 容器嵌入应用程序中。如果使用的是标准的 Spring 配置，则开发人员不会将 Web 容器嵌入应用程序中，而是将其作为 WAR 文件部署到 Web 容器上。这是一个关键的区别，也是 Spring Boot 用于创建部署在微服务架构中的应用程序的最重要原因之一。微服务的一个主要特征是独立于其他微

服务。在这种情况下，很明显它们不应共享公共资源（如数据库或 Web 容器）。由此可见，在一个 Web 容器上部署许多 WAR 文件是微服务所不能接受的模式。因此，Spring Boot 是显而易见的选择。

就个人而言，笔者在开发许多应用程序时都使用了 Spring Boot，而不仅仅是在微服务环境中工作时。如果开发人员尝试使用了 Spring Boot 而不是标准的 Spring Framework 配置，那么笔者相信他将再也不想回过头来使用标准的 Spring Framework 配置。为了支持这个结论，我们找到了一个有趣的图表，它说明了 Java 框架存储库在 GitHub 上的流行程度，其网址为 https://redmonk.com/fryan/2017/06/22/language-framework-popularity-a-look-at-java-june-2017/。开发人员也可以通过网络搜索找到最新的此类统计图表。

接下来我们将具体讨论如何使用 Spring Boot 开发应用程序。

## 2.2　使用 Spring Boot 开发应用程序

要在项目中启用 Spring Boot，我们推荐的方法是使用依赖关系管理系统（Dependency Management System）。在这里，开发人员可以看到如何在 Maven 和 Gradle 项目中包含适当工件的简短片段。以下是 Maven 项目 pom.xml 中的示例片段。

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.5.7.RELEASE</version>
</parent>
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
</dependencies>
```

使用 Gradle，开发人员将不需要定义父依赖项。以下是 build.gradle 中的一个片段。

```
plugins {
    id 'org.springframework.boot' version '1.5.7.RELEASE'
}
dependencies {
    compile("org.springframework.boot:spring-boot-starter-
web:1.5.7.RELEASE")
}
```

在使用 Maven 时，没有必要继承 spring-boot-starter-parent POM。或者，开发人员也可以使用以下依赖关系管理机制。

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-dependencies</artifactId>
            <version>1.5.7.RELEASE</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

现在，开发人员所需要的只是创建主应用程序类并使用@SpringBootApplication 来注解它，这相当于联合使用其他 3 个注解（Annotation），即@Configuration、@EnableAutoConfiguration 和@ComponentScan：

```
@SpringBootApplication
public class Application {

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }

}
```

一旦声明了主类和 spring-boot-starter-web，则开发人员只需要运行第一个应用程序。而且，如果开发人员使用了诸如 Eclipse 或 IntelliJ 之类的集成开发环境（Integrated Development Environment，IDE），则应该只运行主类。否则，必须像标准 Java 应用程序一样使用 java -jar 命令构建和运行应用程序。首先，开发人员应该在应用程序构建期间提供负责将所有依赖项打包到可执行 JAR 的配置。这个可执行 JAR 有时也被称为胖 JAR（Fat JAR）。如果在 Maven 项目的 pom.xml 中定义了 spring-boot-maven-plugin，则会执行该操作。

```
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
```

```
        </plugin>
    </plugins>
</build>
```

该示例应用程序在 Tomcat 容器上仅执行启动 Spring 环境，Tomcat 容器在端口 8080 上可用。胖 JAR 的大小约为 14MB。开发人员可以使用集成开发环境轻松查看项目中包含的库。这些都是基本的 Spring 库，如 spring-core、spring-aop、spring-context，Spring Boot，已经嵌入的 Tomcat，用于日志记录的库如 Logback、Log4j 和 Slf4j，此外还有用于 JSON 的序列化（Serialization）或反序列化（Deserialization）的 Jackson 库。为项目设置默认的 Java 版本是一个很好的选择。开发人员可以通过声明 java.version 属性在 pom.xml 中轻松设置它。

```
<properties>
    <java.version>1.8</java.version>
</properties>
```

开发人员可以通过向 Jetty 服务器添加新的依赖项来更改默认的 Web 容器。

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jetty</artifactId>
</dependency>
```

### 2.2.1　自定义配置文件

快速创建应用程序并且无须很大的工作量，这两者对于开发人员来说其实是一回事，但同样重要的是能够轻松自定义和覆盖默认设置。在这一方面，Spring Boot 可以派上用场，提供支持配置管理的机制。要自定义配置，最简单的方法是使用配置文件，然后将配置文件追加到应用程序胖 JAR。Spring Boot 会自动检测名称以 application 前缀开头的配置文件。它所支持的文件类型是.properties 和.yml。

因此，开发人员可以创建诸如 application.properties 或 application.yml 之类的配置文件，甚至包括特定于配置文件的文件，如 application-prod.properties 或 application-dev.yml。此外，开发人员还可以使用操作系统环境变量和命令行参数来外部化配置。使用 properties 或 YAML 文件时，它们应放在以下位置之一。

- 当前应用程序目录的/config 子目录。
- 当前应用程序目录。
- 类路径/config 包（例如，在 JAR 中）。

- 类路径根。

如果要为配置文件指定一个特定名称，而不是 application 或 application - {profile}，则需要在启动期间提供 spring.config.name 环境属性。开发人员还可以使用 spring.config.location 属性，该属性包含以逗号分隔的目录位置或文件路径列表。

```
java -jar sample-spring-boot-web.jar --spring.config.name=example
java -jar sample-spring-boot-web.jar --
spring.config.location=classpath:/example.properties
```

在配置文件内部，开发人员可以定义两种类型的属性。首先，有一组常见的预定义 Spring Boot 属性，它主要由来自 spring-boot-autoconfigure 库的底层类使用。其次，开发人员还可以考虑定义自己的自定义配置属性，然后使用@ConfigurationProperties 或@Value 注解将其注入应用程序。

现在可以先从预定义的属性开始。Spring Boot 项目所支持的完整列表可以在它们的说明文档的 Common application properties（常见应用程序属性）一节的 Appendix A（附录 A）中得到。其中，大多数都特定于某些 Spring 模块，如数据库、Web 服务器、安全性模块和其他一些解决方案等，但是也有一组核心属性。就个人而言，笔者更喜欢使用 YAML 而不是 properties 文件，因为 YAML 文件更方便阅读，当然，具体使用哪一种文件取决于开发人员的个人喜好。最常见的是覆盖应用程序名称之类的属性（该名称将用于服务发现和分布式配置管理）、网络服务器端口、日志或数据库连接设置。

一般来说，application.yml 文件存放在 src/main/resources 目录中，然后在 Maven 构建之后位于 JAR 根目录中。以下是一个配置文件示例，它将覆盖默认服务器端口、应用程序名称和日志记录属性。

```
server:
    port: ${port:2222}

spring:
    application:
        name: first-service

logging:
    pattern:
        console: "%d{HH:mm:ss.SSS} %-5level %logger{36} - %msg%n"
        file: "%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n"
    level:
        org.springframework.web: DEBUG
    file: app.log
```

这里很值得称道的是，开发人员不必为日志记录配置定义任何其他外部配置文件，如 log4j.xml 或 logback.xml。在之前的文件片段中，可以看到我们已经将 org.springframework.web 的默认日志级别更改为 DEBUG 和日志模式，并创建了一个日志文件 app.log，放置在当前应用程序目录中。现在，默认的应用程序名称是 first-service，默认 HTTP 端口是 2222。

开发人员的自定义配置设置也应放在相同的 properties 或 YAML 文件中。以下是一个带有自定义属性的 application.yml 示例。

```
name: first-service
my:
  servers:
    - dev.bar.com
    - foo.bar.com
```

也可以使用@Value 注解注入一个简单的属性。

```
@Component
public class CustomBean {

    @Value("${name}")
    private String name;

    // ...
}
```

还可以使用@ConfigurationProperties 注解注入更复杂的配置属性。例如，在 YAML 文件中的 my.servers 属性中定义的值列表即可被注入 java.util.List 类型的目标 bean 中。

```
@ConfigurationProperties(prefix="my")
public class Config {

    private List<String> servers = new ArrayList<String>();

    public List<String> getServers() {
        return this.servers;
    }

}
```

到目前为止，我们已经设法创建了一个简单的应用程序，这个程序除了在 Tomcat 或 Jetty 等 Web 容器上启动 Spring 之外什么都没做。本章的这一部分旨在向开发人员展示使用 Spring Boot 启动应用程序开发是多么简单。除此之外，我们还介绍了如何使用 YAML

或 properties 文件自定义配置。对于那些喜欢点击而不是打字的人，我们推荐使用 Spring Initializr 网站（https://start.spring.io/），在该网站可以根据自己选择的选项生成项目存根（Project Stub）。在简单的站点视图中，开发人员可以选择构建工具（Maven/Gradle）、编程语言（Java/Kotlin/Groovy）和 Spring Boot 版本等，然后开发人员应该在 Search for dependencies（搜索依赖项）标签后提供使用该搜索引擎的所有必需的依赖项。我们已经选择了包括 spring-boot-starter-web，它在 Spring Initializr 上标记为 Web，如图 2.1 所示即为该网站的屏幕截图。单击 Generate Project（生成项目）后，包含生成的源代码的 ZIP 文件将下载到开发人员的计算机上。此外，感兴趣的开发人员也可以单击 Switch to the full version（切换到完整版本），这样就可以看到几乎所有可用的 Spring Boot 和 Spring Cloud 库，它们都可以包含在生成的项目中。



图 2.1　Spring Initializr 网站的屏幕截图

既然已经讨论了使用 Spring Boot 构建项目的基础知识，那么现在正是为示例应用程序添加一些新功能的最佳时机。

## 2.2.2　创建 RESTful Web 服务

作为第一步，现在不妨来创建 RESTful Web 服务，向调用客户端公开一些数据。如前所述，负责 JSON 消息序列化和反序列化的 Jackson 库将与 spring-boot-starter-web 一起自动包含在类路径中。鉴于此，开发人员不需要做任何事情，只需声明一个模型类，然后由该模型类通过 REST 方法返回或用作参数。

以下是示例模型类 Person。

```
public class Person {

    private Long id;
    private String firstName;
    private String lastName;
    private int age;
    private Gender gender;

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    //...
}
```

Spring Web 提供了一些可用于创建 RESTful Web 服务的注解。其中的第一个注解是@RestController，该注解应该在负责处理传入 HTTP 请求的控制器 bean 类上设置。此外还有@RequestMapping 注解，它通常用于将控制器方法映射到 HTTP。正如下面的文件片段所示，它可以在整个控制器类上使用，以设置其中所有方法的请求路径。开发人员可以为具体的 HTTP 方法使用更具体的注解，如@GetMapping 或@PostMapping。这里值得注意的是，@GetMapping 与@RequestMapping 相同，它们都具有参数 method = RequestMethod.GET。另外两个常用的注解是@RequestParam 和@RequestBody。@RequestParam 可以绑定路径并查询对象的参数；而@RequestBody 则可以使用 Jackson 库将 JSON 映射到对象。

```
@RestController
@RequestMapping("/person")
public class PersonController {

    private List<Person> persons = new ArrayList<>();

    @GetMapping
    public List<Person> findAll() {
        return persons;
     }

    @GetMapping("/{id}")
```

```
    public Person findById(@RequestParam("id") Long id) {
        return persons.stream().filter(it ->
it.getId().equals(id)).findFirst().get();
    }

    @PostMapping
    public Person add(@RequestBody Person p) {
        p.setId((long) (persons.size()+1));
        persons.add(p);
        return p;
    }

    // ...
}
```

为了与 REST API 标准兼容，开发人员应该处理 PUT 和 DELETE 方法。在其实现之后，开发人员将执行所有创建（Create）、检索（Retrieve）、更新（Update）和删除（Delete）操作，如表 2.2 所示。

表 2.2　GET、POST、PUT 和 DELETE 方法

| 方　法 | 路　径 | 说　明 |
| --- | --- | --- |
| GET | /person | 返回所有现有人员 |
| GET | /person/{id} | 返回具有给定 ID 的人员 |
| POST | /person | 添加新人 |
| PUT | /person | 更新现有人员 |
| DELETE | /person/{id} | 使用给定的 ID 从列表中删除人员 |

以下是使用 DELETE 和 PUT 方法的@RestController 实现示例的代码片段。

```
@DeleteMapping("/{id}")
public void delete(@RequestParam("id") Long id) {
    List<Person> p = persons.stream().filter(it ->
it.getId().equals(id)).collect(Collectors.toList());
    persons.removeAll(p);
}

@PutMapping
public void update(@RequestBody Person p) {
    Person person = persons.stream().filter(it ->
it.getId().equals(p.getId())).findFirst().get();
    persons.set(persons.indexOf(person), p);
}
```

控制器的代码非常简单。它将所有数据都存储在本地 java.util.List 中，这显然不是一个好的编程习惯。当然，这只是为了示例需要而采取的简化的权宜设置。在本章第 2.6 节“将应用程序与数据库集成”一节中，还将详细介绍与 NoSQL 数据库集成的更高级的示例应用程序。

可能部分开发人员有 SOAP Web 服务的开发经验。如果已经使用 SOAP 而不是 REST 创建了类似的服务，则开发人员将为客户端提供一个 WSDL 文件，其中描述了所有服务定义。糟糕的是，REST 并不支持像 WSDL 这样的标准表示法（Notation）。在 RESTful Web 服务的初始阶段，有人说 Web 应用程序描述语言（Web Application Description Language，WADL）将执行该角色。但实际情况是，许多提供商（包括 Spring Web）在应用程序启动后都不会生成 WADL 文件。

当然，这只是一个题外话。简而言之，我们已经完成了第一个微服务，它通过 HTTP 公开了一些 REST 操作。开发人员可能在构建胖 JAR 之后从集成开发环境或使用 java -jar 命令来运行此微服务。如果开发人员没有更改 application.yml 文件中的配置属性，或者在运行应用程序时未设置-Dport 选项，则可以在 http://localhost:2222 下找到它。为了让其他开发人员能够调用我们的 API，这里有两种选择：一种方法是共享描述其用法的文档；另一种方法是采用自动 API 客户端生成的机制，或者也可以两种方法并行，而这正是 Swagger 的用武之地。

## 2.3　API 文档

Swagger 是用于设计、构建和详细说明 RESTful API 的最流行的工具。它由 SmartBear 创建，SmartBear 是一个非常流行的 SOAP Web 服务工具 SoapUI 的设计者。对于那些具有长期使用 SOAP 经验的开发人员来说，这可能是一个很好的建议。无论如何，在使用 Swagger 的情况下，开发人员可以使用表示法设计 API，然后从中生成源代码；或者反过来，可以从源代码开始，然后生成 Swagger 文件。在有了 Spring Boot 之后，开发人员可以考虑使用第二个选项。

### 2.3.1　联合使用 Swagger 2 和 Spring Boot

Spring Boot 和 Swagger 2 之间的集成将由 Springfox 项目实现。它在运行时会检查应用程序，以基于 Spring 配置、类结构和 Java 注解来推断 API 语义。要将 Swagger 与 Spring 结合在一起使用，开发人员需要将以下两个依赖项添加到 Maven 的 pom.xml 中，并且使用@EnableSwagger2 注解主应用程序类。

```
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger2</artifactId>
    <version>2.7.0</version>
</dependency>
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger-ui</artifactId>
    <version>2.7.0</version>
</dependency>
```

在应用程序启动期间，Swagger 库将自动从源代码生成 API 文档。该过程由 Docket bean 控制，该 bean 也将在主类中声明。这里有一个好主意，那就是从 Maven 的 pom.xml 文件中获取 API 版本。要获取该信息，开发人员可以在类路径中包含 maven-model 库并使用 MavenXpp3Reader 类。开发人员还可以使用 apiInfo 方法设置其他一些属性，如标题、作者和详细说明等。默认情况下，Swagger 会为所有 REST 服务生成说明文档，包括由 Spring Boot 创建的服务。开发人员应该将此说明文档仅放置于 pl.piomin.services.boot.controller 包中的@RestController。

```
@Bean
public Docket api() throws IOException, XmlPullParserException {
  MavenXpp3Reader reader = new MavenXpp3Reader();
  Model model = reader.read(new FileReader("pom.xml"));
  ApiInfoBuilder builder = new ApiInfoBuilder()
      .title("Person Service Api Documentation")
      .description("Documentation automatically generated")
      .version(model.getVersion())
      .contact(new Contact("Piotr Mińkowski",
"piotrminkowski.wordpress.com", "piotr.minkowski@gmail.com"));
  return new Docket(DocumentationType.SWAGGER 2).select()
.apis(RequestHandlerSelectors.basePackage("pl.piomin.services.boot.
controller"))
      .paths(PathSelectors.any()).build()
      .apiInfo(builder.build());
}
```

### 2.3.2　使用 Swagger UI 测试 API

在应用程序启动之后，即可在 http://localhost:2222/swagger-ui.html 位置看到 API 说明文档仪表板（Dashboard），这是 Swagger JSON 定义文件的一个对用户更友好的版本，

并且它也是自动生成的，可从 http://localhost:2222/v2/api-docs 获得，如图 2.2 所示。该文件可以由任何其他 REST 工具（如 SoapUI）导入。

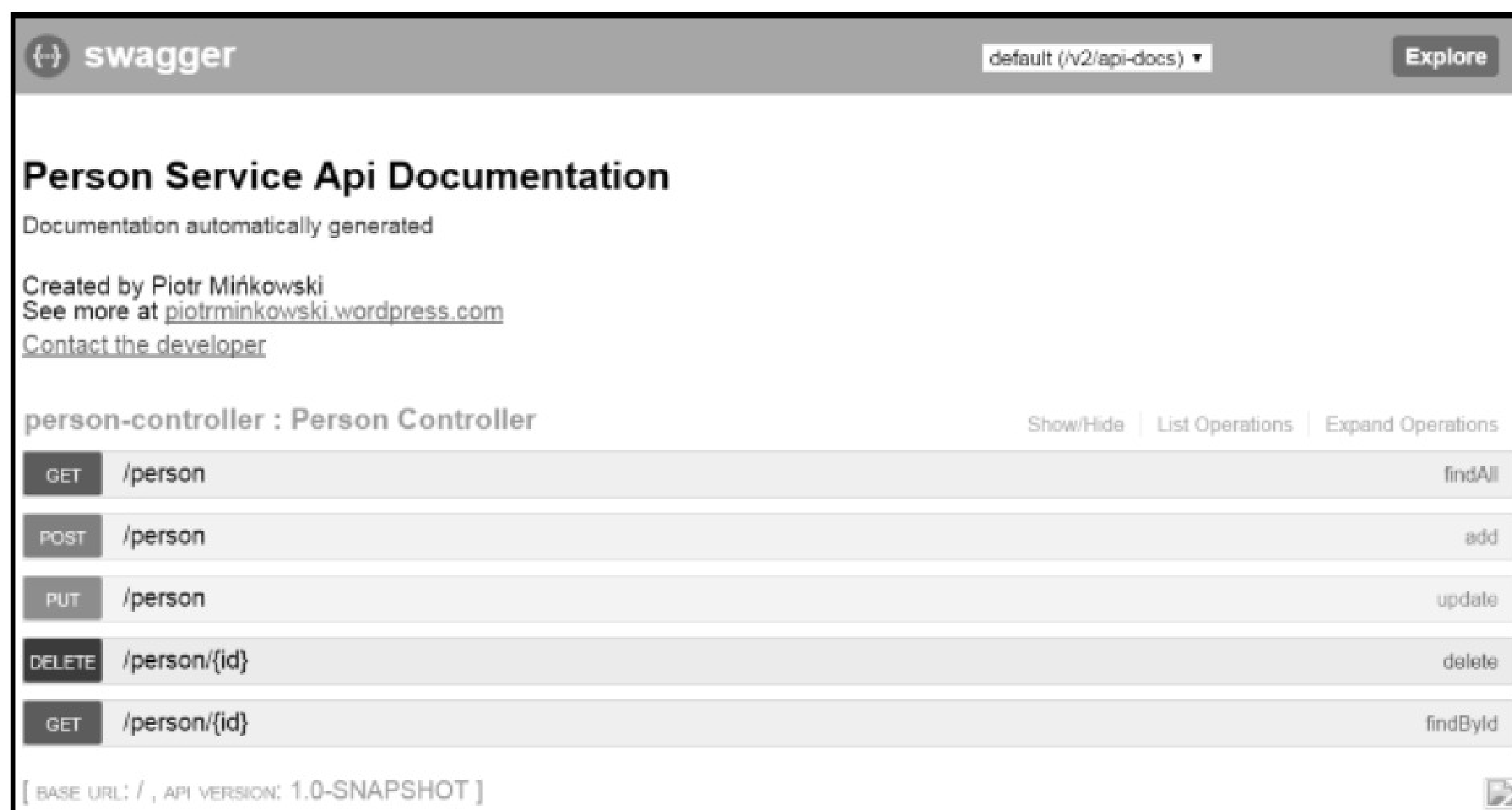



图 2.2　Swagger API 说明文档

如果开发人员更喜欢 SoapUI 而不是 Swagger UI，则可以通过选择 Project | Import Swagger（项目|导入 Swagger）来轻松导入 Swagger 定义文件。然后，需要提供一个文件地址，如图 2.3 所示。

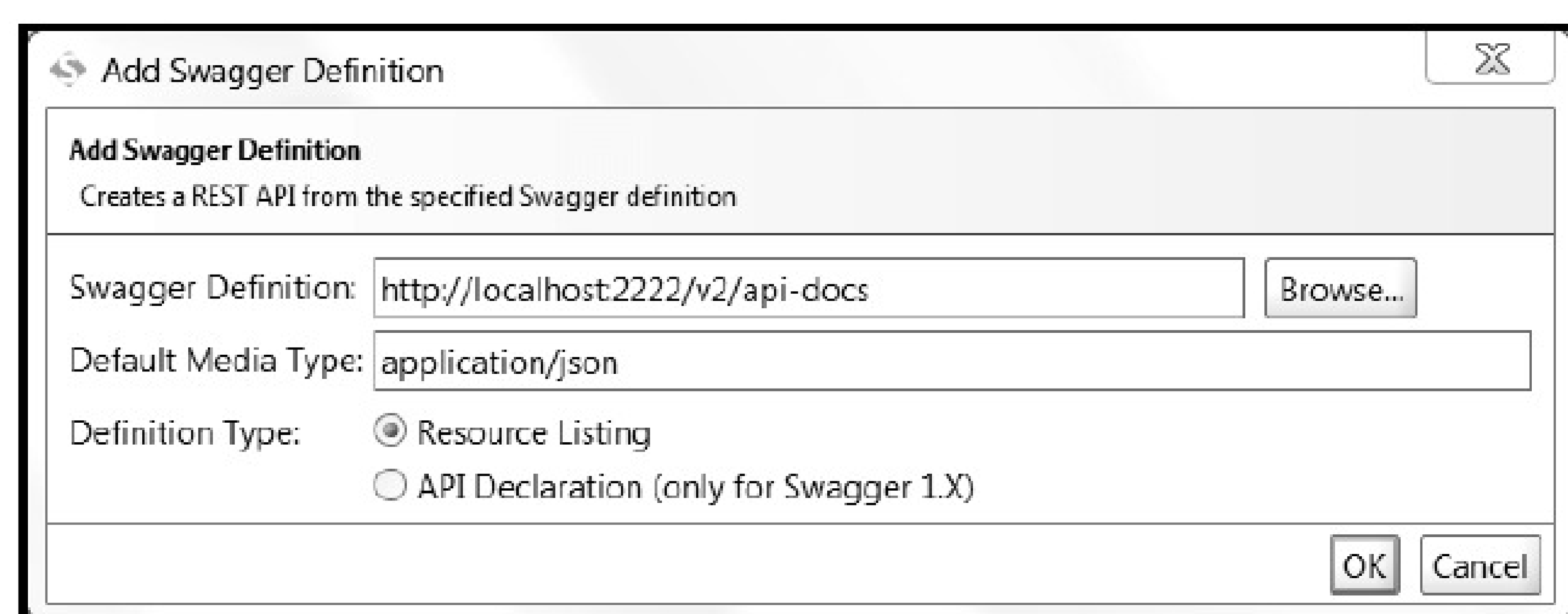



图 2.3　指定要导入的 Swagger 定义文件的位置

就个人而言，笔者更喜欢 Swagger UI。开发人员可以展开每个 API 方法以查看其详细信息。可以通过提供所需的参数或 JSON 输入来测试每个操作，然后单击 Try it Out!（试一试！）按钮。如图 2.4 所示就是发送一个 POST /person 测试请求的截图。

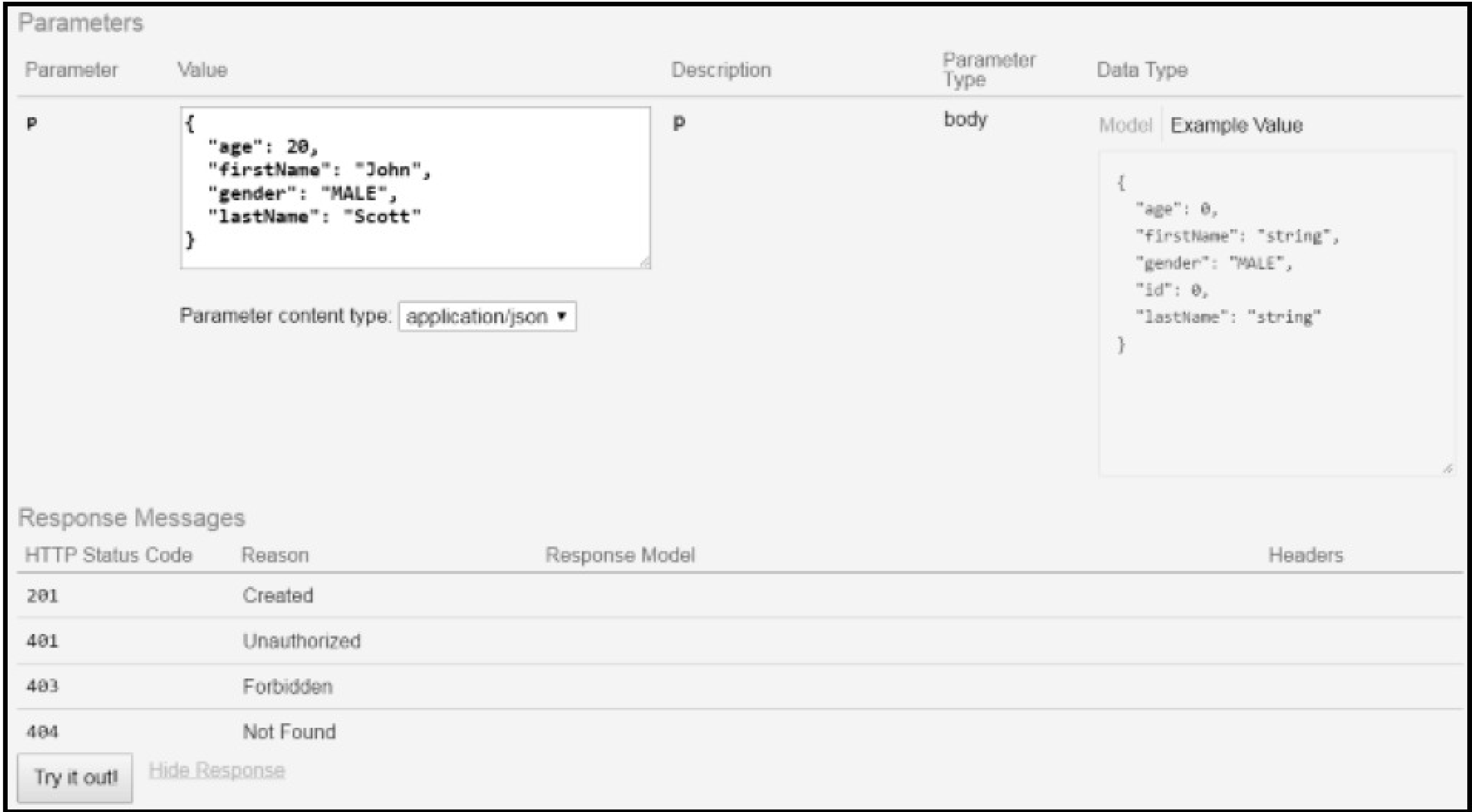



图 2.4　发送 POST /person 测试请求的截图

其响应截图如图 2.5 所示。




图 2.5　POST /person 测试请求的响应截图

## 2.4　Spring Boot 执行器功能

仅创建有效应用程序并共享标准化 API 说明文档并非我们的全部目标，特别是如果我们谈论的是微服务，其中有许多独立实体需要构建一个托管环境。因此，下一个值得一提的重要事项是监控和收集来自应用程序的指标信息。在这方面，Spring Boot 也全程参与。项目 Spring Boot 执行器（Spring Boot Actuator）提供了许多内置端点（Endpoint），允许开发人员监控应用程序并与之交互。要在项目中启用它，开发人员应该在依赖项中包含 spring-boot-starter-actuator。表 2.3 列出了最重要的 Actuator 端点。

表 2.3　重要 Actuator 端点列表

| 路　径 | 说　明 |
| --- | --- |
| /beans | 显示应用程序中初始化的所有 Spring bean 的完整列表 |
| /env | 公开 Spring 的可配置环境中的属性，如操作系统环境变量和配置文件中的属性 |
| /health | 显示应用程序健康信息 |
| /info | 显示任意应用程序信息。例如，可以从 build-info.properties 或 git.properties 文件中获取它 |
| /loggers | 显示和修改应用程序中日志记录器的配置 |
| /metrics | 显示当前应用程序的指标信息，如内存使用情况、正在运行的线程数或 REST 方法响应时间 |
| /trace | 显示跟踪信息（默认情况下为最后 100 个 HTTP 请求） |

可以使用 Spring 配置属性轻松自定义端点。例如，开发人员可以禁用由默认端点启用的其中一个端点。默认情况下，除了用于 shutdown 之外的所有端点都已经启用。这些端点中的大多数都是安全的。如果要从 Web 浏览器中调用它们，则应在请求头（Request Header）中提供安全凭据或禁用整个项目的安全性。要执行后者，开发人员必须在 application.yml 文件中包含以下语句：

```
management:
  security:
    enabled: false
```

### 2.4.1　应用信息

在启动期间，通过应用程序日志可以看到项目可用的完整端点列表。在禁用安全性之后，开发人员可以在 Web 浏览器中测试所有这些内容。有趣的是，/info 端点在默认情

况下并不会提供任何信息。如果开发人员想要更改此设置，则可以使用 3 个可用的自动配置的 InfoContributor bean 之一或编写自己的 bean。在这 3 个 bean 中，第一个 bean 是 EnvironmentInfoContributor，它公开了端点中的环境键值。第二个 bean 是 GitInfoContributor，它将检测类路径中的 git.properties 文件，然后显示有关提交的所有必要信息，如分支名称或提交 ID。最后一个 bean 是 BuildInfoContributor，它可以从 META-INF/build-info.properties 文件中收集信息，并将其显示在端点中。Git 的两个属性文件和构建信息可以在应用程序构建期间自动生成。要实现这一点，开发人员应该在 pom.xml 中包含 git-commit-id-plugin 并自定义 spring-boot-maven-plugin，按以下方式生成 build-info.properties 文件。

```
<plugin>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-maven-plugin</artifactId>
    <executions>
        <execution>
            <goals>
                <goal>build-info</goal>
                <goal>repackage</goal>
            </goals>
            <configuration>
                <additionalProperties>
                    <java.target>${maven.compiler.target}</java.target>
                    <time>${maven.build.timestamp}</time>
                </additionalProperties>
            </configuration>
        </execution>
    </executions>
</plugin>
<plugin>
    <groupId>pl.project13.maven</groupId>
    <artifactId>git-commit-id-plugin</artifactId>
    <configuration>
    <failOnNoGitDirectory>false</failOnNoGitDirectory>
    </configuration>
</plugin>
```

在 build-info.properties 文件可用的情况下，开发人员的/info 将与以前略有不同。

```
{
    "build": {
        "version":"1.0-SNAPSHOT",
        "java": {
```

```
            "target":"1.8"
        },
        "artifact":"sample-spring-boot-web",
        "name":"sample-spring-boot-web",
        "group":"pl.piomin.services",
        "time":"2017-10-04T10:23:22Z"
    }
}
```

## 2.4.2　健康信息

与/info 端点一样，/health 端点也有一些自动配置的指示器。开发人员可以监控磁盘使用情况、邮件服务、Java 消息服务（Java Message Service，JMS）、数据源和 NoSQL 数据库（如 MongoDB 或 Cassandra）的状态等。如果从我们的示例应用程序中检查该端点，则只能获得有关磁盘使用情况的信息。现在我们可以将 MongoDB 添加到项目中，以测试一个可用的健康指标 MongoHealthIndicator。MongoDB 不是随机选择的，它对于我们未来的 Person 微服务的更高级示例是有用的。要启用 MongoDB，开发人员需要将以下依赖项添加到 pom.xml。

de.flapdoodle.embed.mongo 工件将负责在应用程序启动期间启动嵌入式数据库的实例。

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-mongodb</artifactId>
</dependency>
<dependency>
    <groupId>de.flapdoodle.embed</groupId>
    <artifactId>de.flapdoodle.embed.mongo</artifactId>
</dependency>
```

现在，/health 端点将返回有关磁盘的使用情况和 MongoDB 状态的信息。

```
{
    "status":"UP",
    "diskSpace":{
        "status":"UP",
        "total":499808989184,
        "free":193956904960,
        "threshold":10485760
    },
    "mongo":{
```

```
        "status":"UP",
        "version":"3.2.2"
    }
}
```

在这个例子中，开发人员可以看到 Spring Boot 自动配置的强大功能。除了为项目添加两个依赖项以启用嵌入式 MongoDB 之外，开发人员不需要做任何事情。其状态已经自动添加到/health 端点。它还有一个可立即使用的 Mongo 客户端连接，可以由存储库 bean 进一步使用。

### 2.4.3 指标信息

众所周知，天上不会掉馅饼，所以，开发快速且简单不可能没有代价，在项目中包含一些额外的库之后，胖 JAR 文件现在大约有 30MB。使用其中一个自动配置的执行器端点/metrics，开发人员可以轻松检查出自己的微服务堆（Heap）和非堆（Non-Heap）内存的使用情况。发送一些测试请求之后，堆的使用量约为 140MB，非堆的使用量则为 65MB。该应用程序的总内存使用量约为 320MB。当然，即使只是在使用 java -jar 命令启动期间使用-Xmx 参数，这些值也可以减少一点。但是，如果我们关心生产模式下的可靠工作，则不应该过多地减少这个限制值。除内存使用外，/metrics 端点还将显示有关已加载类的数量、活动线程数、每种 API 方法的平均持续时间等信息。以下是我们的示例微服务的端点响应的片段。

```
{
    "mem":325484,
    "mem.free":121745,
    "processors":4,
    "instance.uptime":765785,
    "uptime":775049,
    "heap.committed":260608,
    "heap.init":131072,
    "heap.used":138862,
    "heap":1846272,
    "nonheap.committed":75264,
    "nonheap.init":2496,
    "nonheap.used":64876,
    "threads.peak":28,
    "threads.totalStarted":33,
    "threads":28,
    "classes":9535,
```

```
    "classes.loaded":9535,
    "gauge.response.person":7.0,
    "counter.status.200.person":4,
    // ...
}
```

开发人员可以创建自己的自定义指标。Spring Boot Actuator 提供了两个类，以方便想要这样做的开发人员：CounterService 和 GaugeService。

CounterService，字面意思是“计数器服务”，顾名思义，它公开了值的递增、递减和重置的方法。相比之下，GaugeService 仅用于提交当前值。API 方法调用统计信息的默认指标有点不完善，因为它们仅基于调用路径。如果方法类型在同一路径上可用，则无法区分它们。在我们的示例端点中，这适用于 GET /person、POST /person 和 PUT /person。无论如何，笔者已经创建了 PersonCounterService bean，它可以统计 add 和 delete 方法调用的次数。

```
@Service
public class PersonCounterService {
    private final CounterService counterService;

    @Autowired
    public PersonCounterService(CounterService counterService) {
        this.counterService = counterService;
    }

    public void countNewPersons() {
        this.counterService.increment("services.person.add");
    }

    public void countDeletedPersons() {
        this.counterService.increment("services.person.deleted");
    }
}
```

需要将此 bean 注入我们的 REST 控制器 bean，并且可以在添加或删除人员时调用递增计数器值的方法。

```
public class PersonController {

    @Autowired
    PersonCounterService counterService;

    // ...
```

```
    @PostMapping
    public Person add(@RequestBody Person p) {
        p.setId((long) (persons.size()+1));
        persons.add(p);
        counterService.countNewPersons();
        return p;
    }

    @DeleteMapping("/{id}")
    public void delete(@RequestParam("id") Long id) {
        List<Person> p = persons.stream().filter(it ->
it.getId().equals(id)).collect(Collectors.toList());
        persons.removeAll(p);
        counterService.countDeletedPersons();
    }
}
```

现在，如果再次显示应用程序的指标信息，则开发人员将在 JSON 响应中看到以下两个新的字段。

```
{
    // ...
    "counter.services.person.add":4,
    "counter.services.person.deleted":3
}
```

Spring Boot 应用程序生成的所有指标信息都可以从内存缓冲区中导出，然后放到可以分析和显示它们的位置。例如，开发人员可以将它们存储在 Redis、Open TSDB、Statsd 甚至 InfluxDB 中。

以上就是本章想要告诉开发人员的关于内置监视器端点的所有细节。本节已经为说明文档、指标和运行状况检查等主题指定了相对大量的空间，在我们看来，这些是微服务开发和维护的重要方面。开发人员通常不关心这些机制是否能得到很好的实现，但其他人通常只是通过这些指标、运行健康状况检查和应用程序的日志质量等反映出来的现象来了解我们的应用程序。Spring Boot 提供了一种现成可用的实现，因此开发人员不必花费太多时间即可启用它们。

## 2.5　开发者工具

Spring Boot 为开发人员提供了一些其他有用的工具。在我们看来，非常好的是，只

要项目类路径上的文件发生变化，应用程序就会自动重启。如果开发人员使用 Eclipse 作为集成开发环境，那么启用它的唯一方法是将 spring-boot-devtools 依赖项添加到 Maven 的 pom.xml 中。然后，尝试更改其中一个类中的某些内容并保存，这样应用程序就会自动重启，并且它所花的时间比标准方式停止和启动要少得多。当启动上述示例应用程序时，大约需要 9 秒，而自动重启仅需 3 秒。

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
    <optional>true</optional>
</dependency>
```

如果在发生更改时不需要触发重启，则可以排除某些资源。默认情况下，指向文件夹的类路径上可用的任何文件的更改都将会被纳入监控，甚至还包括静态资源或视图模板，但是这些都不需要重新启动。例如，如果将它们放在静态文件夹中，则可以通过将以下属性添加到 application.yml 配置文件来排除它们。

```
spring:
 devtools:
  restart:
   exclude: static/**
```

## 2.6　将应用程序与数据库集成

在 Spring Boot 规范中还可以找到更多有趣的功能。虽然笔者很想花更多的时间来描述该框架提供的其他很好的功能，但还是不应该偏离本章的主题——使用微服务的 Spring。在前文中已经提过，通过在项目中包含嵌入式 MongoDB，本节将向开发人员提供一个更高级的微服务示例。在开始具体的讨论之前，不妨回顾一下当前版本的应用程序。它的源代码可以在笔者的公共 GitHub 账户上找到。开发人员可以将以下 Git 存储库克隆到本地计算机。

https://github.com/piomin/sample-spring-boot-web.git。

基本示例在 master 分支中可用。包含嵌入式 MongoDB 的更高级示例将提交给 mongo 分支。如果开发人员想尝试运行更高级的示例，则需要使用 git checkout mongo 切换到该分支。现在，我们需要在模型类中执行一些更改，以启用映射到 MongoDB 的对象。模型类必须使用@Document 注解，主键字段（Primary Key Field）则使用@Id 注解。本示例还将 ID 字段类型从 Long 更改为 String，因为 MongoDB 将会以 UUID 格式生成主键，如

59d63385206b6d14b854a45c。

```
@Document(collection = "person")
public class Person {

    @Id
    private String id;

    private String firstName;
    private String lastName;
    private int age;
    private Gender gender;

    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }
    // ...
}
```

下一步是创建一个扩展 MongoRepository 的存储库接口。MongoRepository 提供了搜索和存储数据的基本方法，如 findAll、findOne、save 和 delete。Spring Data 有一个非常智能的机制，用于使用存储库对象执行查询。开发人员不必自己实现查询，只要定义具有正确名称的接口方法即可。该方法的名称应具有前缀 findBy，然后是搜索的字段名称。它可能以标准搜索关键字后缀结束，如 GreaterThan、LessThan、Between、Like 等。Spring Data 类将基于完整的方法名称自动生成 MongoDB 查询。

相同的关键字也可以与 delete…By 或 remove…By 一起使用，以创建删除查询。在 PersonRepository 接口中，可以定义两个查找方法。其中，第一个是 findByLastName，它将选择具有给定 lastName 值的所有 Person 实体；第二个是 findByAgeGreaterThan，它将检索年龄大于给定值的所有 Person 实体。

```
public interface PersonRepository extends MongoRepository<Person, String> {

    public List<Person> findByLastName(String lastName);
    public List<Person> findByAgeGreaterThan(int age);

}
```

应将存储库注入 REST 控制器类。然后，开发人员最终可以调用 PersonRepository 提供的所有必需的 CRUD 方法。

```
@Autowired
private PersonRepository repository;
@Autowired
private PersonCounterService counterService;

@GetMapping

public List<Person> findAll() {
    return repository.findAll();
}

@GetMapping("/{id}")
public Person findById(@RequestParam("id") String id) {
    return repository.findOne(id);
}

@PostMapping
public Person add(@RequestBody Person p) {
    p = repository.save(p);
    counterService.countNewPersons();
    return p;
}

@DeleteMapping("/{id}")
public void delete(@RequestParam("id") String id) {
    repository.delete(id);
    counterService.countDeletedPersons();
}
```

我们还为 PersonRepository bean 中的自定义查找操作添加了两个 API 方法。

```
@GetMapping("/lastname/{lastName}")
public List<Person> findByLastName(@RequestParam("lastName") String
lastName) {
    return repository.findByLastName(lastName);
}
```

```
@GetMapping("/age/{age}")
public List<Person> findByAgeGreaterThan(@RequestParam("age") int age) {
    return repository.findByAgeGreaterThan(age);
}
```

以上就是本示例必须要做的一切。我们的微服务公开了在嵌入的 Mongo 数据库上实现 CRUD 操作的基本 API 方法，并且它已准备好启动。读者可能已经注意到，它并不需要我们创建大量源代码。使用 Spring Data 实现与数据库（无论是关系数据库还是 NoSQL）的任何交互都是快速且相对容易的。除此之外，我们还面临着另一个挑战。虽然嵌入式数据库是一个不错的选择，但它仅限于开发模式或单元测试，而不是在生产环境中。如果必须在生产模式（Production Mode）下运行微服务，则可能会启动一个独立实例或部署为分片集群（Sharded Cluster）的 Mongo 实例，并将应用程序连接到它们。出于本示例目的，我们将使用 Docker 运行 MongoDB 的单个实例。

**注意：**

如果开发人员不熟悉 Docker，则可以在本地或远程计算机上安装 Mongo。有关 Docker 的更多信息，开发人员还可以参考本书第 14 章“Docker 支持”，其中包含了对于 Docker 的简要介绍。在该章中，开发人员将找到所需的一切，例如，如何在 Windows 上安装它并使用基本命令。为了后续章节的讨论主题，我们还将在示例实现中使用 Docker，所以，如果开发人员掌握了与 Docker 相关的基础知识，那么将会方便很多。

## 2.7　运行应用程序

现在来使用 Docker 的 `run` 命令启动 MongoDB。

```
docker run -d --name mongo -p 27017:27017 mongo
```

可能对开发人员很有用的东西是 Mongo 数据库客户端。使用此功能，可以创建新数据库并为某些用户添加凭据。如果在 Windows 上安装了 Docker，则其默认的虚拟机地址为 192.168.99.100。由于在 run 命令中设置了-p 参数，所以 Mongo 容器的端口 27017 已公开。事实上，开发人员不必创建数据库，因为在定义客户端连接时需要提供数据库的名称，如果该数据库不存在，那么它将自动创建一个，如图 2.6 所示。

接下来，应该为具有足够权限的应用程序创建用户，如图 2.7 所示。

Add Connection

Connection　Authentication　URL　SSH

Connection Name　mongo_docker

Hostname　192.168.99.100

Port　27017

DB Name　microservices

Read From SECONDARY　For replica sets, legacy slaveOK=true

Close　Save changes

图 2.6　在定义客户端连接时需要提供数据库的名称

Add User

User　micro

Password　•••••

Roles　Custom Data

Show 10 entries　Search:

| Role | Database | Delete |
| --- | --- | --- |
| dbOwner | microservices | ✖ |

Showing 1 to 1 of 1 entries　Previous 1 Next

Add Role

Close　Save

图 2.7　添加用户

最后，应该在 application.yml 配置文件中设置 Mongo 数据库连接设置和凭据。

```
server:
  port: ${port:2222}
spring:
  application:
  name: first-service

// ...

---

spring:
  profiles: production
  application:
    name: first-service
  data:
    mongodb:
      host: 192.168.99.100
      port: 27017
      database: microservices
      username: micro
      password: micro
```

Spring Boot 对多配置文件配置有很好的支持。可以使用“---”线将 YAML 文件分隔为一系列文档，并对文档的每个部分进行独立解析。前面所介绍的示例与带有 application-production.yml 的分离配置文件完全相同。如果在没有任何其他选项的情况下运行该应用程序，那么它将使用默认设置，而默认设置是没有配置文件名称设置的。如果要使用生产属性运行它，则应设置 VM 参数 spring.profiles.active。

```
java -jar -Dspring.profiles.active=production sample-spring-boot-web-1.0-SNAPSHOT.jar
```

这还不是全部。现在，具有活动生产配置文件的应用程序无法启动，因为它会尝试初始化 embeddedMongoServer bean。开发人员可能已经知道，绝大多数其他解决方案会在 Spring Boot 中设置了自动配置，在这种情况下也没有什么不同。所以，我们还需要在生产配置文件中排除自动配置中的 EmbeddedMongoAutoConfiguration 类。

```
spring:
  profiles: production
  // ...
  autoconfigure:
```

```
    exclude:
org.springframework.boot.autoconfigure.mongo.embedded.
EmbeddedMongoAutoConfiguration
```

也可以使用配置类来排除该工件。

```
@Configuration
@Profile("production")
@EnableAutoConfiguration(exclude = EmbeddedMongoAutoConfiguration.class)
public class ApplicationConfig {
    // ...
}
```

当然，开发人员还可以使用更简洁的解决方案，如使用 Maven 配置文件，并且从目标构建包（Target Build Package）中排除整个 de.flapdoodle.embed.mongo 工件。这里所提出的解决方案只是解决问题的几种可能方式之一，但它显示了 Spring Boot 中的自动配置和配置文件机制。现在，开发人员可以运行该示例应用程序并使用诸如 Swagger UI 之类的用户界面来执行一些测试。还可以使用 Mongo 客户端连接到数据库，并检查数据库中的更改。以下是示例项目的最终文件结构。

```
pl
  +- piomin
    +- services
      +- boot
        +- Application.java
        |
        +- controller
        |   +- PersonController.java
        |
        +- data
        |   +- PersonRepository.java
        |
        +- model
        |   +- Person.java
        |   +- Gender.java
        |
        +- service
        |   +- PersonCounterService.java
```

示例应用程序已经完成。这些都是本章意在向开发人员展示的 Spring Boot 功能。我们所关注的正是那些对创建基于 REST 的服务特别有用的东西。

## 2.8 小　　结

本章已经引导开发人员完成单一微服务开发的过程，从一个非常基础的示例过渡到一个更高级的已经可用于生产模式的 Spring Boot 应用程序。本章详细介绍了如何使用启动器为项目启用其他功能；使用 Spring Web 库实现公开 REST API 方法的服务；并且讨论了使用属性和 YAML 文件自定义服务配置，解释了如何为已公开的 REST 端点提供详细说明文档和规范。此外，本章还配置了运行状况检查和健康功能，使用了 Spring Boot 配置文件来调整应用程序以不同的模式运行。最后，本章还使用 ORM 功能实现了与嵌入式和远程 NoSQL 数据库的交互。

本章没有提到与 Spring Cloud 有关的任何内容，这并非偶然。如果没有使用 Spring Boot 的基本知识和经验，开发人员就无法开始使用 Spring Cloud 项目。Spring Cloud 提供了许多不同的功能，允许开发人员将服务放在基于微服务的完整生态系统中。我们将在接下来的章节中逐一讨论这些功能。

# 第 3 章　Spring Cloud 概述

在本书第 1 章“微服务简介”中，提到了云原生的开发风格，并且 Spring Cloud 可以帮助开发人员轻松采用与此概念相关的最佳实践。实际上，在一个名为 The Twelve-Factor App（微服务十二要素）的趣味倡议中已经收集了一些最常用的最佳实践。该倡议 App 的访问地址为 https://12factor.net/，中文版地址为 https://12factor.net/zh_cn/。在该倡议中开宗明义地提出，软件通常会作为一种服务来交付，它们被称为网络应用程序，或软件即服务（Software as a Service，SaaS）。The Twelve-Factor App 为构建 SaaS 应用提供了方法论，它提出现代应用程序必须可扩展、可在云平台上轻松部署，并且可以按连续部署的方式交付。开发人员有必要熟悉这些原则，特别是，如果开发人员所构建的应用程序将作为服务运行的话，则更应该熟悉它们。Spring Boot 和 Spring Cloud 提供的功能和组件将使得开发人员的应用程序可以符合十二要素规则（Twelve-Factor Rules）。开发人员可以区分最现代的分布式系统通常使用的一些典型特征。每一个遵守十二要素规则的框架都应该提供它们，Spring Cloud 也不例外。

本章将要讨论的主题包括：

- 分布式/版本化配置。
- 服务注册和发现。
- 路由。
- 服务和服务之间的调用。
- 负载均衡。
- 断路器。
- 分布式消息传递。

## 3.1　从基础开始

在本书第 2 章中，曾详细介绍了 Spring Boot 项目的结构。开发人员应在 YAML 或 properties 文件中提供配置，并且应包含应用程序或 application-{profile}名称。与标准的 Spring Boot 应用程序相比，Spring Cloud 基于从远程服务器获取的配置。但是，应用程序内部仅需要最少的设置，如它的名称和配置服务器地址。这就是 Spring Cloud 应用程序需要创建 Bootstrap 上下文（Bootstrap Context）的原因，这个 Bootstrap 上下文将负责从

外部源加载属性。

Bootstrap 属性以最高优先级添加，并且本地配置无法覆盖它们。Bootstrap 上下文是主应用程序上下文的父级，它将使用 bootstrap.yml 而不是 application.yml。一般来说，可以将应用程序名称和 Spring Cloud Config 设置如下。

```
spring:
  application:
    name: person-service
  cloud:
    config:
     uri: http://192.168.99.100:8888
```

通过将 spring.cloud.bootstrap.enabled 属性设置为 false，开发人员可以轻松禁用 Bootstrap 上下文启动。此外，还可以使用 spring.cloud.bootstrap.name 属性更改 Bootstrap 引导程序配置文件的名称，甚至可以通过设置 spring.cloud.bootstrap.location 来更改其位置。因为这里也提供了配置文件机制，所以开发人员还可以创建诸如 bootstrap-development.yml 之类的文件，并将其加载到活动的开发配置文件（Development Profile）中。Spring Cloud Context 库中提供了此功能和其他一些功能，它将作为父依赖项（Parent Dependency）与任何其他 Spring Cloud 库一起添加到项目类路径中。在这些功能中，有一项是 Spring Boot Actuator 附带的一些额外管理端点。

- env：它是针对 Environment 的新 POST 方法，将执行日志级别的更新，并且重新绑定@ConfigurationProperties。
- refresh：它将重新加载 Bootstrap 上下文，并且刷新所有使用@RefreshScope 注解的 bean。
- restart：它将重新启动 Spring ApplicationContext。
- pause：它将停止 Spring ApplicationContext。
- resume：它将启动 Spring ApplicationContext。

和 Spring Cloud Context 一起使用的下一个库是 Spring Cloud Commons，它同样作为父依赖项包含在 Spring Cloud 项目中。它将为诸如服务发现、负载均衡和断路器之类的机制提供一个通用的抽象层，其中还包括一些常用的注解，如@EnableDiscoveryClient 或@LoadBalanced 等。后续章节将介绍有关它们的详细信息。

### 3.1.1　Netflix OSS

在阅读本书前两章时，开发人员可能已经注意到许多与微服务架构相关的关键字的出现。对于部分开发人员来说，这可能是一个新名词，而对于其他人来说，这是众所周

知的。但是到目前为止，本书尚未提及微服务网络社区的一个重要词汇。大多数人都知道，这个词就是 Netflix（美国最大的在线 DVD 租赁商，中文名：奈飞公司）。很多人都喜欢该公司提供的电视节目和其他作品，但是对于开发人员来说，它是因其他原因而闻名的，这个原因就是微服务。Netflix 是最早从创建单一的一体化应用程序的传统开发模式迁移到云原生微服务开发方法的先驱之一。该公司通过将大部分源代码推送到公共存储库、在会议演示中发言以及发布博客帖文等方式来与网络社区分享其专业经验。Netflix 架构的概念非常成功，它也成为其他大型企业的榜样，而它的 IT 架构师（如 Adrian Cockcroft 等）现在则是微服务的杰出传播者。此后，许多开源框架都基于 Netflix 共享的代码提供了它们的解决方案库。Spring Cloud 也不例外，它提供了与最流行的 Netflix OSS 功能的集成，这些功能包括 Eureka、Hystrix、Ribbon 和 Zuul 等。

顺便说一句，不知道你是否一直关注 Netflix，但他们对自己决定开放大部分源代码的原因做出了清晰的阐述，我认为这值得引用，因为这也部分地解释了他们的解决方案在 IT 世界中大获成功和广受欢迎的原因：

> “当我们说要将 Netflix 全部移植到云端时，每个人都说我们完全疯了。他们不相信我们实际上已经这样做了，他们认为我们只会把事情搞砸。”

### 3.1.2 使用 Eureka 进行服务发现

Spring Cloud Netflix 提供的第一个模式是使用 Eureka 的服务发现。该软件包分为客户端和服务器两部分。

要在项目中包含 Eureka 客户端，开发人员应该使用相应的 spring-cloud-starter-eureka 启动器。客户端始终是应用程序的一部分，负责连接到远程发现服务器。一旦建立起连接，它应该发送带有服务名称和网络位置的注册消息。如果当前微服务必须从另一个微服务调用端点，则客户端应该从服务器检索具有已注册服务列表的最新配置。服务器可以作为独立的 Spring Boot 应用程序进行配置和运行，并且应该设定为高度可用，每个服务器都可以将其状态复制到其他节点。要在项目中包含 Eureka Server，开发人员需要使用 spring-cloud-starter-eureka-server 启动器。

### 3.1.3 使用 Zuul 路由

在 Spring Cloud Netflix 项目下可用的下一个流行模式是使用 Zuul 进行智能路由。它不仅是基于 JVM 的路由器，而且还可以充当服务器端负载均衡器或执行一些过滤。它还

可以具有各种各样的应用。Netflix 将其用于身份验证、减轻负载、静态响应处理或压力测试等情况。它与 Eureka Server 的相同之处在于，它可以作为独立的 Spring Boot 应用程序进行配置和运行。

要在项目中包含 Zuul，需要使用 spring-cloud-starter-zuul 启动器。在微服务架构中，Zuul 扮演着 API 网关的重要角色，它是整个系统的入口点。它需要了解每个服务的网络位置，因此它可以通过将发现客户端（Discovery Client）包含到类路径中来与 Eureka Server 进行交互。

### 3.1.4　使用 Ribbon 实现负载均衡

Spring Cloud Netflix 的下一个功能同样是开发人员不能忽视的，因为它就是用于实现客户端负载均衡的 Ribbon。Ribbon 支持最流行的协议，如 TCP、UDP 和 HTTP 等。它不仅可以用于同步 REST 调用，还可以用于异步和反应模型。除了负载均衡之外，它还能提供与服务发现、缓存、批处理和容错功能等的集成。Ribbon 是建立在基本 HTTP 和 TCP 客户端之上的新抽象层次。

要将 Ribbon 包含在项目中，需要使用 spring-cloud-starter-ribbon 启动器。Ribbon 支持轮询调度（Round Robin）算法、可用性过滤和现成可用的加权响应时间负载均衡规则（Weighted Response Time Load Balancing Rule），并且可以使用自定义规则对其进行轻松扩展。它将基于命名客户端（Named Client）概念，这意味着应该为包含负载均衡的服务器提供一个名称。

### 3.1.5　编写 Java HTTP 客户端

Feign 是一个受欢迎程度略低的 Netflix OSS 软件包。它是一个声明性 REST 客户端，可以帮助开发人员更轻松地编写 Web 服务客户端。使用 Feign 之后，开发人员只需要声明和注解接口，而实际的实现将在运行时生成。

要在项目中包含 Feign，需要使用 spring-cloud-starter-feign 启动器。它可以与 Ribbon 客户端集成，因此在默认情况下就已经支持负载均衡和其他 Ribbon 功能，包括与发现服务的通信。

### 3.1.6　Hystrix 的延迟和容错能力

在本书第 1 章“微服务简介”中已经提到了断路器模式，Spring Cloud 提供了一个实现这种模式的库。它基于由 Netflix 创建的作为断路器实现的 Hystrix 软件包。默认情况下，

Hystrix 将与 Ribbon 和 Feign 客户端集成在一起。回退（Fallback）与断路器概念密切相关。使用 Spring Cloud 库，开发人员可以轻松配置回退逻辑，如果存在读取或断路器超时，则应执行该回退逻辑。要在项目中包含 Hystrix，需要使用 spring-cloud-starter-hystrix 启动器。

### 3.1.7　使用 Archaius 进行配置管理

Spring Cloud Netflix 项目提供的最后一个重要功能是 Archaius。就个人而言，笔者尚未使用过这个库，但它在某些情况下可能会很有用。Spring Cloud 参考资料中称 Archaius 是 Apache Commons Configuration 项目的扩展。它允许通过轮询（Polling）源的更改或将更改推送到客户端来更新配置。

## 3.2　发现和分布式配置

服务发现和分布式配置管理是微服务架构的重要组成部分。这两种不同机制的技术实现非常相似。它可以归结为将特定键下的参数存储在灵活的键-值存储中。实际上，市场上有若干种有趣的解决方案都可以提供这两种功能。Spring Cloud 集成了它们之中最受欢迎的产品，但是还有一个例外的地方，那就是 Spring Cloud 只为分布式配置创建了自己的实现。此功能在 Spring Cloud Config 项目下可用。相比之下，Spring Cloud 没有为服务注册和发现提供自己的实现。

像往常一样，我们可以将此项目划分为服务器和客户端支持。服务器是一个中心位置，可以在所有环境中管理应用程序的所有外部属性。它可以在多个版本和配置文件中同时维护配置，这是通过使用 Git 作为存储后端来实现的。该机制非常智能，本书第 5 章“使用 Spring Cloud Config 进行分布式配置”中对此有详细讨论。Git 后端不是存储属性的唯一选项。配置文件也可以位于文件系统或服务器类路径上。下一个选项是使用 Vault 作为后端。Vault 是一个开源工具，用于管理 HashiCorp 发布的令牌、密码或证书等机密。我们知道，许多组织都特别关注安全问题，例如，将凭证存储在安全的地方，因此，它可能是适合这些组织的解决方案。一般来说，开发人员还可以管理配置服务器访问级别的安全性。无论哪个后端用于存储属性，Spring Cloud Config Server 都会公开一个基于资源的 HTTP API，通过该 API 可以轻松访问它们。默认情况下，该 API 将使用基本身份验证进行保护，但也可以使用私钥/公钥身份验证设置 SSL 连接。

服务器可以作为独立的 Spring Boot 应用程序运行，其属性通过 REST API 公开。要

为项目启用它，开发人员应该添加 spring-cloud-config-server 依赖项。客户端也有支持。在创建任何 Spring bean 之前，每个使用配置服务器作为属性源的微服务都需要在启动之后立即连接到它。有趣的是，非 Spring 应用程序也可以使用 Spring Cloud Config Server。有一些流行的微服务框架会在客户端与它集成。要为应用程序启用 Spring Cloud Config Client，开发人员需要包含 spring-cloud-config-starter 依赖项。

### 3.2.1　可选替代方案——Consul

Netflix 发现和 Spring 分布式配置的一个有趣的替代方案似乎是由 Hashicorp 创建的 Consul。Spring Cloud 提供了与这一流行工具的集成，以便用于在基础架构中的发现和配置服务。像往常一样，开发人员可以使用一些简单的通用注解来启用该集成。与之前提供的解决方案相比，唯一的区别在于配置设置。为了与 Consul 服务器建立通信，其代理（Agent）需要可用于应用程序。它必须能够作为一个独立的进程运行，默认情况下，它应该在 http://localhost:8500 地址处可用。此外，Consul 还提供了 REST API，该 API 可以直接用于注册、收集服务列表或属性配置。

要激活 Consul Service Discovery，需要使用 spring-cloud-starter-consul-discovery 启动器。在应用程序启动和注册之后，客户端应该会查询 Consul 以查找其他服务。它支持以下两种功能：使用 Netflix Ribbon 的客户端负载均衡器，以及使用 Netflix Zuul 的动态路由器和过滤器。

### 3.2.2　Apache Zookeeper

Spring Cloud 支持的该领域下的一个流行的解决方案是 Apache Zookeeper。根据其文档说明，它是用于维护配置、命名的集中式服务，该服务还提供分布式同步，并且能够对服务进行分组。之前适用于 Consul 的有关 Spring Cloud 支持的所有内容对于 Zookeeper 来说也同样适用。这里值得一提的是简单的通用注解，它在很多情况下都必须用到，例如，启用集成、通过设置文件中的属性进行配置，以及用于与 Ribbon 或 Zuul 交互的自动配置。

要在客户端使用 Zookeeper 启用服务发现，不仅需要包含 spring-cloud-starter-zookeeper-discovery，还需要包含 Apache Curator，后者提供了一个 API 框架和实用程序，使集成变得简单和可靠。分布式配置客户端不需要它，开发人员只需要使项目依赖项包含 spring-cloud-starter-zookeeper-config 即可。

### 3.2.3　其他项目

值得一提的还有另外两个项目，它们现在正处于孵化阶段。所有这些项目都可以在 GitHub 存储库中找到，其地址为 https://github.com/spring-cloud-incubator。其中一些可能会在短期内正式附加到 Spring Cloud 软件包。第一个要说的是 Spring Cloud Kubernetes，它提供了与这个非常流行的工具的集成。关于它我们有很多可以讲述的东西，但为了简短起见，这里不妨就用几句话来概括。它是一个可以对容器化应用程序（Containerized Application）进行自动部署、扩展和管理的系统，最初由 Google 设计；它可用于容器编排，并具有许多有趣的功能，包括服务发现、配置管理和负载均衡等。在某些情况下，它可能会被视为 Spring Cloud 的竞争对手。其配置将随 YAML 文件的使用一起提供。

Spring Cloud 的重要特性是服务发现和分布式配置机制，这些机制都可以在 Kubernetes 平台上获得。要在应用程序中使用它们，开发人员需要包含 spring-cloud-starter-kubernetes 启动器。

第二个值得一提的处于孵化阶段的有趣项目是 Spring Cloud Etcd。与上述项目一样，其主要功能是分布式配置、服务注册和发现。Etcd 不是像 Kubernetes 这样强大的工具，它只是提供了一个分布式键-值存储，它具有在集群环境中存储数据的可靠方法。此外，Etcd 还是 Kubernetes 中服务发现、集群状态和配置管理的后端。

## 3.3　使用 Sleuth 进行分布式跟踪

Spring Cloud 的另一个基本功能是分布式跟踪（Distributed Tracing），它可以在 Spring Cloud Sleuth 库中实现。其主要目的是在处理单个输入请求时，关联在不同微服务之间分派的后续请求。与大多数情况一样，这些是基于 HTTP 头实现跟踪机制的 HTTP 请求。该实现是在 Slf4j 和 MDC 上构建的。Slf4j 为特定的日志框架（如 logback、log4j 或 java.util.logging）提供了外观和抽象。映射诊断上下文（Mapped Diagnostic Context，MDC）是用于区分不同来源的日志输出的解决方案，该解决方案还可以使用实际范围中不可用的其他信息来丰富日志输出。

Spring Cloud Sleuth 可以将跟踪和跨度（Span）ID 添加到 Slf4J MDC 中，以便开发人员能够提取具有给定跟踪或跨度的所有日志。它还添加了一些其他条目，如应用程序名称或可导出标志。它集成了最流行的消息传递解决方案（如 Spring REST 模板、Feign 客户端、Zuul 过滤器、Hystrix 或 Spring Integration 消息通道等）。它也可以与 RxJava 或计

划任务一起使用。

要在项目中启用 Spring Cloud Sleuth，需要添加 spring-cloud-starter-sleuth 依赖项。对于开发人员来说，基本跨度和跟踪 ID 机制的使用是完全透明的。

添加跟踪头并不是 Spring Cloud Sleuth 的唯一功能。它还负责记录计时信息，这在延迟分析中很有用。这些统计数据可以导出到 Zipkin，后者是一种可用于查询和可视化计时数据的工具。

**注意：**

Zipkin 是一种分布式跟踪系统，专门用于分析微服务架构中的延迟问题。它公开了用于收集输入数据的 HTTP 端点。要为 Zipkin 生成和发送跟踪，应该将 spring-cloud-starter-zipkin 依赖项包含在项目中。

一般来说，没有必要分析一切。鉴于输入的流量非常高，因此，开发人员只需要收集一定比例的数据。基于这个目的，Spring Cloud Sleuth 提供了一个采样策略，开发人员可以决定向 Zipkin 发送多少输入的流量。解决大数据问题的第二个智能解决方案是使用消息代理（Message Broker）发送统计信息，而不是使用默认的 HTTP 端点。要启用此功能，必须包含 spring-cloud-sleuth-stream 依赖项，它允许应用程序成为消息的生产者，并且发送到 Apache Kafka 或 RabbitMQ。

## 3.4　消息传递和集成

前文已经提到了消息传递代理及其在应用程序和 Zipkin 服务器之间进行通信的用法。一般来说，Spring Cloud 可以通过同步/异步 HTTP 和消息传递代理支持两种类型的通信。该领域的第一个项目是 Spring Cloud Bus，它允许开发人员向应用程序发送广播事件，将有关状态的修改（如配置属性更新或其他管理命令等）通知给应用程序。实际上，开发人员也可能希望使用具有 RabbitMQ 代理的 AMQP 启动器或 Apache Kafka 的启动器，其启用方法和前文所述是一样的，只需要将 spring-cloud-starter-bus-amqp 或 spring-cloud-starter-bus-kafka 包含在依赖关系管理中即可，所有其他必要的操作都将通过自动配置来执行。

Spring Cloud Bus 是一个相当小的项目，它允许开发人员使用分布式消息传递功能进行常见操作，如广播配置更改事件。但是，如果要构建由消息驱动的微服务组成的系统，那么正确的框架选择应该是 Spring Cloud Stream。这是一个非常强大的框架，也是最大的 Spring Cloud 项目之一，本书专门用了一个整章（第 11 章“消息驱动的微服务”）来详

细介绍它。与 Spring Cloud Bus 相同，它有两个绑定器（Binder）可用，第一个用于 AMQP 和 RabbitMQ，第二个则用于 Apache Kafka。Spring Cloud Stream 基于 Spring Integration（这是 Spring 的另一个大型项目），它提供了一种编程模型，支持大多数企业集成模式（Enterprise Integration Pattern），如端点、通道、聚合器（Aggregator）或转换器（Transformer）等。整个微服务系统中包含的应用程序将通过 Spring Cloud Stream 输入和输出通道进行相互通信。它们之间的主要通信模型是发布/订阅（Publish/Subscribe），其中的消息将通过共享主题广播。此外，支持每个微服务的多个实例也很重要。在大多数情况下，消息应仅由单个实例处理，而发布/订阅模型不支持单个实例，这就是为什么 Spring Cloud Stream 引入了分组机制，在该机制中，只有一个组成员从目的地接收消息。正如前文所述，这两个启动器也可以包含在项目中，具体包含哪一个则取决于绑定器的类型：spring-cloud-starter-stream-kafka 或 spring-cloud-starter-stream-rabbit。

还有两个与 Spring Cloud Stream 相关的项目。第一个是 Spring Cloud Stream App Starters，它定义了一组 Spring Cloud Stream 应用程序，这些应用程序都可以独立运行或使用马上要介绍的第二个项目 Spring Cloud Data Flow。在这些应用程序中，开发人员可以区分连接器、网络协议适配器和通用协议。Spring Cloud Data Flow 是另一个应用广泛且又功能强大的 Spring Cloud 工具包。它通过为构建数据集成和实时数据处理管道提供智能解决方案，简化了开发和部署。基于微服务的数据管道的编排是通过简单的 DSL、拖放式用户界面仪表板和 REST API 联合完成的。

## 3.5　云平台支持

Pivotal Cloud Foundry（PCF）是一个用于部署和管理现代应用程序的云原生平台。有些开发人员可能已经知道，Pivotal Software 公司是 Spring framework 商标的所有者。大型商业平台的赞助是 Spring 越来越受欢迎的重要原因之一。显而易见的是，PCF 完全支持 Spring Boot 的可执行 JAR 文件，以及所有 Spring Cloud 微服务模式（如 Config Server、服务注册表和断路器等）。这些类型的工具都可以轻松运行和配置，并且可以使用各种用户界面仪表板或客户端命令行。PCF 的开发甚至比标准的 Spring Cloud 应用程序更简单。开发人员唯一要做的就是将正确的启动器包含在项目依赖项中。

- spring-cloud-services-starter-circuit-breaker
- spring-cloud-services-starter-config-client
- spring-cloud-services-starter-service-registry

很难找到一个没有亚马逊云服务（Amazon Web Services，AWS）支持的特立独行的

云架构。Spring Cloud 也不例外。Spring Cloud for Amazon Web Services 提供了与最流行的 Web 工具的集成，这包括用于与简单队列服务（Simple Queueing Service，SQS）、简单通知服务（Simple Notification Service，SNS）、ElasticCache 和关系数据库服务（Relational Database Service，RDS）进行通信的模块。其中，RDS 提供了诸如 Aurora、MySQL 或 Oracle 之类的引擎。可以使用 CloudFormation 堆栈中定义的名称访问远程资源。众所周知的 Spring 常规和模式中的一切都是不透明的，它有 4 个主要模块可用。

- Spring Cloud AWS Core：包括使用 spring-cloud-starter-aws 启动器，提供可直接访问 EC2 实例的核心组件。
- Spring Cloud AWS Context：提供对简单存储服务（Simple Storage Service）、简单电子邮件服务（Simple E-mail Service）和缓存服务的访问。
- Spring Cloud AWS JDBC：包括使用 starter spring-cloud-starter-aws-jdbc 启动器，提供数据源查找和配置功能，可与 Spring 支持的任何数据访问技术一起使用。
- Spring Cloud AWS Messaging：包括使用 starter spring-cloud-starter-aws-messaging 启动器，允许应用程序使用 SQS（点对点）或 SNS（发布/订阅）发送和接收消息。

还有一个值得一提的项目，尽管它仍处于发展的早期阶段，这就是 Spring Cloud Function，它可以为无服务器（Serverless）架构提供支持。这里所谓的“无服务器”也称为功能即服务（Function-as-a-Service，FaaS），开发人员只需要创建非常小的模块，这些模块部署在完全由第三方提供商管理的容器上。实际上，Spring Cloud Functions 为 AWS Lambda 和 Apache OpenWhisk（最受欢迎的 FaaS 提供商）实现了适配程序。我们将持续关注这个旨在支持无服务器方法的项目的开发。

在本节中，不应遗漏的还包括 Spring Cloud Connectors 项目（以前称为 Spring Cloud）。它为部署在云平台上的基于 JVM 的应用程序提供了抽象层。实际上，它支持 Heroku 和 Cloud Foundry，在该项目中，开发人员的应用程序可以使用 Spring Cloud Heroku Connectors 和 Spring Cloud Foundry Connector 其中一种模块连接 SMTP、RabbitMQ、Redis 或其中一个可用的关系数据库。

## 3.6　其他有用的库

围绕微服务架构还存在一些重要的考量，虽然这些方面不能被视为其核心功能，但也非常重要。第一个要考虑的就是安全性。

### 3.6.1 安全性

在 Spring Security 和 Spring Web 项目中提供了使用 OAuth2、JSON Web 令牌（JSON Web Token，JWT）或基本身份验证等机制保护 API 安全的标准实现的重要部分。Spring Cloud Security 使用这些库来允许开发人员轻松创建实现常见模式的系统，如单点登录（Single Sign-on）和令牌中继（Token Relay）。要为应用程序启用安全管理，应该包含 spring-cloud-starter-security 启动器。

### 3.6.2 自动化测试

关于微服务开发的下一个要考虑的重要方面是自动化测试。对于微服务架构来说，契约测试（Contract Test）的重要性日益增加。ThoughtWorks 公司的首席科学家 Martin Fowler 给出了以下定义：

“集成契约测试是在外部服务边界进行的测试，它将用于验证其是否符合消费服务所期望的契约。”

Spring Cloud 对于单元测试方法 Spring Cloud Contract 有一个非常有趣的现象。它使用 WireMock 进行流量记录，并使用 Maven 插件生成存根。

开发人员也有机会使用 Spring Cloud Task。它可以帮助开发人员使用 Spring Cloud 创建短期活跃的微服务，并在本地或云环境中运行它们。要在项目中启用它，应该包含 spring-cloud-starter-task 启动器。

### 3.6.3 集群功能

最后要考虑的一个重要方面（也是最后一个项目）是 Spring Cloud Cluster。它为领导选举和常见的状态模式提供了解决方案，并为 Zookeeper、Redis、Hazelcast 和 Consul 提供了抽象和实现。

## 3.7 项 目 概 述

如前文所述，Spring Cloud 包含许多子项目，可提供与许多不同工具和解决方案的集

成。这很容易让人感到茫然（因为要梳理和了解的东西太多），如果开发人员首次使用 Spring Cloud 则更是如此。一个直观的图表可能胜过千言万语，为此，图 3.1 以分门别类的方式呈现了 Spring Cloud 最重要的项目。

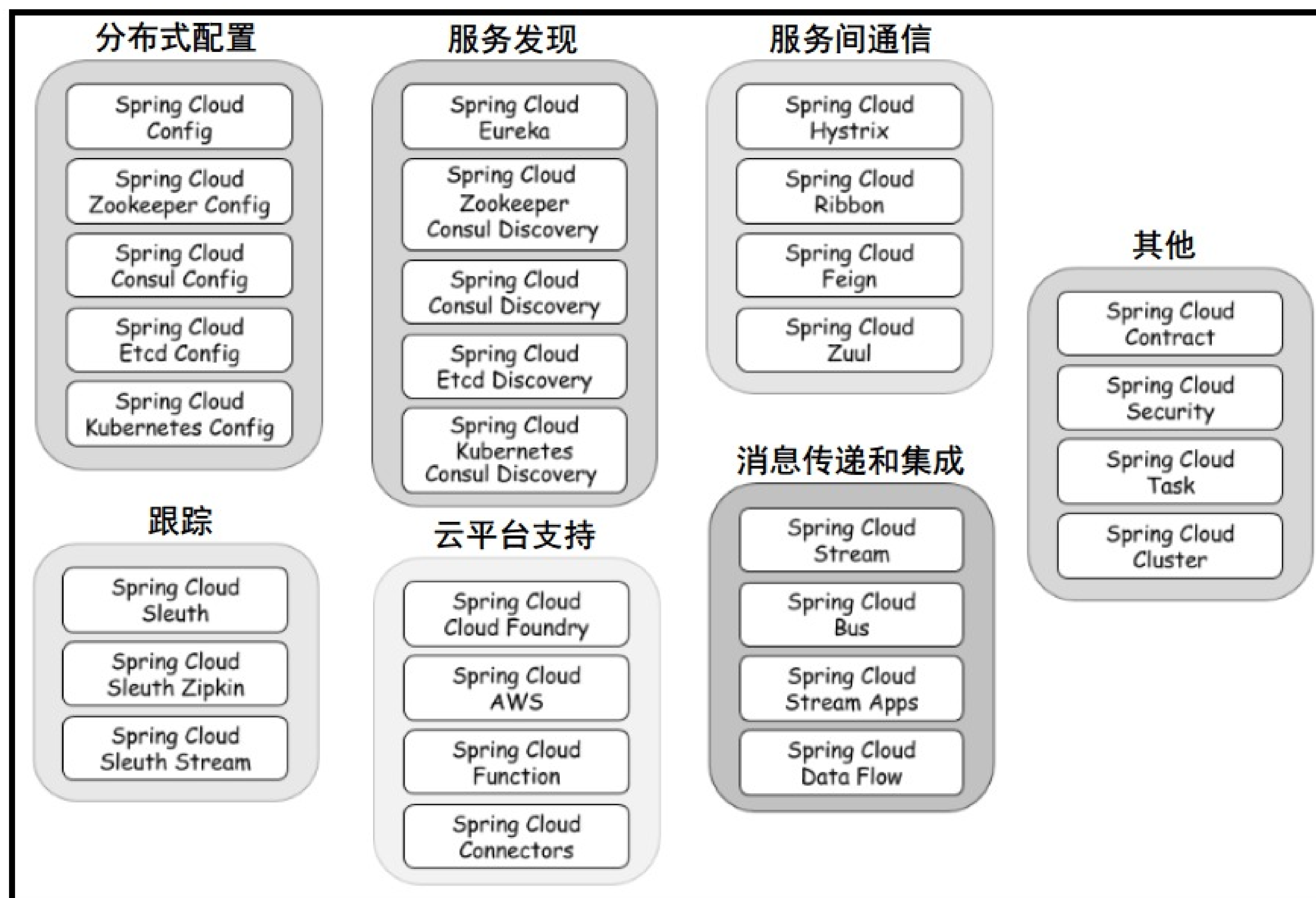



图 3.1　Spring Cloud 项目分类

## 3.8　版 本 列 车

正如图 3.1 所示，Spring Cloud 中有许多项目，它们之间有很多关系。根据定义，这些都是具有不同版本层叠和版本号的独立项目。在这种情况下，开发人员的应用程序中的依赖关系管理可能会出现问题，并且需要提供关于所有项目版本之间关系的知识。为了简化操作，Spring Cloud 引入了启动器机制（前文已经介绍过）和版本列车（Release Train）。版本列车将通过名称而不是版本来识别，以避免与子项目产生混淆。

有趣的是，版本列车以伦敦地铁站命名，按字母顺序排列。第一个版本是 Angel，第二个版本是 Brixton，依此类推。

依赖项管理的整个机制基于物料清单（Bill of Materials，BOM），这是用于管理独立版本工件的标准 Maven 概念。以下是一个实际表格，其中包含的 Spring Cloud 项目版

本已经被分配给版本列车。有些名称带有后缀 M[X]，其中的[X]是版本号，M[X]则表示里程碑（Milestone），说明它是稳定可靠的重要版本。SR[X]表示服务版本（Service Release），意指该版本修复了一些关键错误。正如表 3.1 所示，Spring Cloud Stream 拥有自己的版本列车，它使用与 Spring Cloud 项目相同的规则对其子项目进行分组。

**表 3.1　Spring Cloud 版本列车**

| 组　件 | Camden.SR7 | Dalston.SR4 | Edgware.M1 | Finchley.M2 | Finchley.BUILD-SNAPSHOT |
|---|---|---|---|---|---|
| spring-cloud-aws | 1.1.4.RELEASE | 1.2.1.RELEASE | 1.2.1.RELEASE | 2.0.0.M1 | 2.0.0.BUILD-SNAPSHOT |
| spring-cloud-bus | 1.2.2.RELEASE | 1.3.1.RELEASE | 1.3.1.RELEASE | 2.0.0.M1 | 2.0.0.BUILD-SNAPSHOT |
| spring-cloud-cli | 1.2.4.RELEASE | 1.3.4.RELEASE | 1.4.0.M1 | 2.0.0.M1 | 2.0.0.BUILD-SNAPSHOT |
| spring-cloud-commons | 1.1.9.RELEASE | 1.2.4.RELEASE | 1.3.0.M1 | 2.0.0.M2 | 2.0.0.BUILD-SNAPSHOT |
| spring-cloud-contract | 1.0.5.RELEASE | 1.1.4.RELEASE | 1.2.0.M1 | 2.0.0.M2 | 2.0.0.BUILD-SNAPSHOT |
| spring-cloud-config | 1.2.3.RELEASE | 1.3.3.RELEASE | 1.4.0.M1 | 2.0.0.M2 | 2.0.0.BUILD-SNAPSHOT |
| spring-cloud-netflix | 1.2.7.RELEASE | 1.3.5.RELEASE | 1.4.0.M1 | 2.0.0.M2 | 2.0.0.BUILD-SNAPSHOT |
| spring-cloud-security | 1.1.4.RELEASE | 1.2.1.RELEASE | 1.2.1.RELEASE | 2.0.0.M1 | 2.0.0.BUILD-SNAPSHOT |
| spring-cloud-cloudfoundry | 1.0.1.RELEASE | 1.1.0.RELEASE | 1.1.0.RELEASE | 2.0.0.M1 | 2.0.0.BUILD-SNAPSHOT |
| spring-cloud-consul | 1.1.4.RELEASE | 1.2.1.RELEASE | 1.2.1.RELEASE | 2.0.0.M1 | 2.0.0.BUILD-SNAPSHOT |
| spring-cloud-sleuth | 1.1.3.RELEASE | 1.2.5.RELEASE | 1.3.0.M1 | 2.0.0.M2 | 2.0.0.BUILD-SNAPSHOT |
| spring-cloud-stream | Brooklyn.SR3 | Chelsea.SR2 | Ditmars.M2 | Elmhurst.M1 | Elmhurst.BUILD-SNAPSHOT |
| spring-cloud-zookeeper | 1.0.4.RELEASE | 1.1.2.RELEASE | 1.2.0.M1 | 2.0.0.M1 | 2.0.0.BUILD-SNAPSHOT |
| spring-boot | 1.4.5.RELEASE | 1.5.4.RELEASE | 1.5.6.RELEASE | 2.0.0.M3 | 2.0.0.M3 |
| spring-cloud-task | 1.0.3.RELEASE | 1.1.2.RELEASE | 1.2.0.RELEASE | 2.0.0.M1 | 2.0.0.RELEASE |

现在，开发人员需要做的就是在 Maven 的 pom.xml 的依赖关系管理部分提供正确的版本列车名称，然后使用启动器包含项目。

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>Finchley.M2</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
<dependencies>
```

```
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-config</artifactId>
    </dependency>
    ...
</dependencies>
```

以下是 Gradle 的相同示例。

```
dependencyManagement {
    imports {
        mavenBom ':spring-cloud-dependencies:Finchley.M2'
    }
}
dependencies {
    compile ':spring-cloud-starter-config'
    ...
}
```

## 3.9 小　　结

本章介绍了 Spring Cloud 中最重要的项目，它们都是 Spring Cloud 的组成部分。此外，本章还对每个项目进行了明确的分类，阅读完本章之后，相信开发人员应该能够清晰地知道应用程序中必须包含哪个库才能实现服务发现、分布式配置、断路器或负载均衡器等模式。开发人员还应该认识到应用程序上下文和 Bootstrap 上下文之间的差异，并了解如何使用基于“版本列车”概念的依赖关系管理在项目中包含依赖项。本章需要引起开发人员注意的最后一件事是与 Spring Cloud 集成的一些工具，如 Consul、Zookeeper、RabbitMQ 或 Zipkin。本章对这些工具做出了不同程度的介绍，并指出了负责与这些工具交互的项目。

本章完成了本书的第一部分。在这一部分中，主要目标是让开发人员了解与 Spring Cloud 项目相关的基础知识。阅读完本部分之后，开发人员应该能够识别基于微服务的架构中最重要的元素，能有效地使用 Spring Boot 来创建简单和更高级的微服务，最后，开发人员还应该能够列出所有最受欢迎的子项目，它们也是 Spring Cloud 的组成部分。

现在，开发人员可以继续阅读本书的下一部分，详细了解那些负责在 Spring Cloud 中实现分布式系统的常见模式的子项目。其中大多数都基于 Netflix OSS 库。后续内容将从提供服务注册表、Eureka 发现服务器的解决方案开始。

# 第二部分

## 微服务架构常见元素和 Spring Cloud 实现

# 第4章　服务发现

在本章之前已经多次讨论过服务发现。实际上，它是微服务架构中最受欢迎的技术方面之一。Netflix OSS 实现中不能省略这样的主题，他们并没有决定使用具有类似功能的任何现有工具，而是专门为自己的需求设计和开发了一个发现服务器。然后，它已经与其他几个工具一起开源。Netflix OSS 发现服务器被称为 Eureka。

用于与 Eureka 集成的 Spring Cloud 库由两部分组成，即客户端和服务器端。服务器将作为单独的 Spring Boot 应用程序启动，并公开一个 API，该 API 将允许收集已注册服务的列表，以及添加带有位置地址的新服务。可以配置和部署服务器以使其具有高可用性，每个服务器都将其状态复制到其他服务器。客户端作为依赖项包含在微服务应用程序中，它负责启动后的注册、关闭前的注销，以及通过轮询 Eureka Server 使注册列表保持最新。

本章将要讨论的主题包括：

- 开发运行嵌入式 Eureka Server 的应用程序。
- 从客户端应用程序连接到 Eureka Server。
- 高级发现客户端配置。
- 在客户端和服务器之间启用安全通信。
- 配置故障转移和对等复制机制。
- 在不同区域中注册客户端应用程序的实例。

## 4.1　在服务器端运行 Eureka

在 Spring Boot 应用程序中运行 Eureka Server 并不是一项很难的任务。现在不妨来看一看如何做到这一点。

（1）首先，必须将正确的依赖项包含在开发人员的项目中。显然，这里需要使用相应的启动器。

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-eureka-server</artifactId>
</dependency>
```

（2）还应在主应用程序类上启用 Eureka Server。

```
@SpringBootApplication
@EnableEurekaServer
public class DiscoveryApplication {

    public static void main(String[] args) {
        new
SpringApplicationBuilder(DiscoveryApplication.class).web(true).run(
args);
    }

}
```

（3）有趣的是，客户端的依赖项将与服务器的启动器一起被包含，它们对开发人员是很有用的，但仅限于在高可用性模式下启动 Eureka 并在发现实例之间进行对等（Peer-to-Peer）通信。在运行独立实例时，除了在启动期间在日志中打印一些错误之外，它实际上并没有其他用处。开发人员可以从初始依赖项中排除 spring-cloud-netflix-eureka-client，也可以考虑使用配置属性禁用发现客户端。建议开发人员优先选择第二种方式，并且在这种情况下，可以将默认服务器端口更改为 8080 以外的其他内容。以下就是 application.yml 文件的代码片段。

```
server:
 port: ${PORT:8761}
eureka:
 client:
   registerWithEureka: false
   fetchRegistry: false
```

（4）在完成上述步骤之后，开发人员即可启动第一个 Spring Cloud 应用程序。可以从集成开发环境运行 main 类或使用 Maven 构建项目。使用 java -jar 命令运行它并等待出现日志行 Started Eureka Server。启动完成之后，会出现一个简单的用户界面仪表板，它可以用作主页，地址是 http://localhost:8761，并且可以使用/eureka/*路径调用 HTTP API 方法。Eureka 仪表板不会提供太多功能，事实上，它主要用于检查注册服务列表。这可以通过调用 REST API http://localhost:8761/eureka/apps 端点来找到。

总而言之，开发人员已经知道如何使用 Spring Boot 运行 Eureka 独立服务器，以及如何使用用户界面控制台和 HTTP 方法检查已注册微服务的列表。但是，目前开发人员仍然没有能够在发现中注册自己的任何服务，现在是时候改变它了。具有发现服务器和客户端实现的示例应用程序可以在 master 分支中的 GitHub（https://github.com/piomin/

sample-spring-cloud-netflix.git）上获得。

## 4.2　在客户端启用 Eureka

与服务器端一样，只需要包含一个依赖项即可为应用程序启用 Eureka Client。因此，首先要在项目的依赖项中包含以下启动器。

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-eureka</artifactId>
</dependency>
```

该示例应用程序只是与 Eureka Server 进行通信。它必须注册自己并发送元数据（Metadata）信息，如主机、端口、运行状况指示器 URL 和主页等。Eureka 将从属于服务的每个实例接收心跳（Heartbeat）消息。如果在已配置的一段时间后未收到心跳消息，则会从注册表中删除该实例。发现客户端的第二个职责是从服务器获取数据，然后缓存它并定期请求更新。可以通过使用@EnableDiscoveryClient 注解 main 类来启用它。令人惊讶的是，还有另一种方法可以激活此功能。开发人员可以使用@EnableEurekaClient 注解，尤其是如果在类路径中有多个发现客户端实现（如 Consul、Eureka、ZooKeeper）时更是如此。前面提到的@EnableDiscoveryClient 注解存在于 spring-cloud-commons 中，而@EnableEurekaClient 注解则存在于 spring-cloud-netflix 中，并且仅适用于 Eureka。以下是发现客户端应用程序的 main 类。

```
@SpringBootApplication
@EnableDiscoveryClient
public class ClientApplication {

    public static void main(String[] args) {
        new
SpringApplicationBuilder(ClientApplication.class).web(true).run(args);
    }

}
```

不必在客户端的配置中提供发现服务器地址，因为它在默认主机和端口上可用。但是，开发人员可以很容易地想象 Eureka 没有侦听其默认的 8761 端口。配置文件的片段如下所示。可以使用 EUREKA_URL 参数覆盖发现服务器的网络地址，客户端的侦听端口也可以使用 PORT 属性覆盖。应用程序在发现服务器中注册的名称将取自 spring.application.name

属性。

```
spring:
 application:
   name: client-service

server:
 port: ${PORT:8081}

eureka:
 client:
   serviceUrl:
     defaultZone: ${EUREKA_URL:http://localhost:8761/eureka/}
```

现在可以在 localhost 上运行示例客户端应用程序的两个独立实例。要实现这一点，应该在启动时为实例覆盖侦听端口的数量，如下所示。

```
java -jar -DPORT=8081 target/sample-client-service-1.0-SNAPSHOT.jar
java -jar -DPORT=8082 target/sample-client-service-1.0-SNAPSHOT.jar
```

如图 4.1 所示，有两个 client-service（客户端服务）实例注册了主机名为 piomin，端口则分别为 8081 和 8082。

Instances currently registered with Eureka

| Application | AMIs | Availability Zones | Status |
|---|---|---|---|
| CLIENT-SERVICE | n/a (2) | (2) | UP (2) - piomin:client-service:8081 , piomin:client-service:8082 |

General Info

| Name | Value |
|---|---|
| total-avail-memory | 322mb |
| environment | test |
| num-of-cpus | 4 |
| current-memory-usage | 68mb (21%) |
| server-uptime | 00:00 |
| registered-replicas | http://localhost:8761/eureka/ |
| unavailable-replicas | http://localhost:8761/eureka/, |
| available-replicas | |

Instance Info

| Name | Value |
|---|---|
| ipAddr | 192.168.1.101 |
| status | UP |

图 4.1　使用 Eureka 注册的两个实例

## 4.2.1　关机时取消注册

检查撤销注册如何与 Eureka 客户端一起工作是一项艰巨的任务。开发人员的应用程

序应该正常关闭，以便能够拦截已停止的事件（Event）并将事件发送到服务器。正常关闭的最佳方法是使用 Spring Actuator 的/shutdown 端点。该执行器是 Spring Boot 的一部分，它可以通过在 pom.xml 中声明 spring-boot-starter-actuator 依赖项来包含在项目中。默认情况下它是被禁用的，因此必须在配置属性中启用它。为简单起见，可以禁用该端点的用户/密码安全性。

```
endpoints:
 shutdown:
   enabled: true
   sensitive: false
```

要关闭应用程序，必须调用 POST /shutdown API 方法。如果收到回复{"message": "Shutting down,bye ..."}，则表示一切顺利，进程已经启动。在禁用应用程序之前，将打印出从 Shutting down DiscoveryClient ...（正在关闭 DiscoveryClient）行开始的一些日志。之后，该服务将从发现服务器取消注册，并从注册服务列表中完全消失。可以通过调用 http://localhost:8082/shutdown 来关闭客户端实例#2（可以使用任何 REST 客户端来调用它，如 Postman），因此，现在只有在端口 8081 上运行的实例仍然可以在仪表板中看到，如图 4.2 所示。

Instances currently registered with Eureka

| Application | AMIs | Availability Zones | Status |
|---|---|---|---|
| CLIENT-SERVICE | n/a (1) | (1) | UP (1) - piomin:client-service:8081 |

图 4.2　现在只有一个实例

Eureka Server 仪表板还提供了一种方便的方法来检查新创建和已取消租约（Lease）的历史记录，如图 4.3 所示。

Last 1000 cancelled leases　Last 1000 newly registered leases

| Timestamp | Lease |
|---|---|
| Nov 1, 2017 11:36:11 PM | CLIENT-SERVICE(piomin:client-service:8082) |
| Nov 1, 2017 11:36:11 PM | CLIENT-SERVICE(piomin:client-service:8082) |

图 4.3　查看已取消租约的历史记录

虽然像这样从容地关闭显然是最适合停止应用程序的方法，但在现实世界中，开发人员并不总是能够实现它。可能会发生许多意外情况，如服务器计算机重新启动、应用程序故障或客户端与服务器之间的接口处的网络问题。从发现服务器的角度来看，从集成开发环境停止客户端应用程序或从命令行终止进程，其情形没什么不同。如果开发人

员尝试这样做，则将看到不会触发发现客户端的关闭进程，并且在具有 UP 状态的 Eureka 仪表板中仍然可以看到该服务。此外，该租约将永不过期。

为了避免这种情况，应该更改服务器端的默认配置。为什么这样的问题会出现在默认设置中呢？实际上，这是因为 Eureka 提供了一种特殊机制，当注册表检测到一定数量的服务没有及时续订它们的租约时，注册表将停止过期条目。这应该可以防止注册表在发生网络故障时清除所有条目。该机制称为自我保护模式（Self-Preservation Mode），开发人员可以使用 application.yml 中的 enableSelfPreservation 属性禁用它。当然，在实际生产模式中它不应该被禁用。

```
eureka:
 server:
   enableSelfPreservation: false
```

## 4.2.2　以编程方式使用发现客户端

客户端应用程序启动后，将自动从 Eureka Server 获取已注册服务的列表。但是，这可能需要以编程方式使用 Eureka 的客户端 API。开发人员有以下两种可能的选择。

- ❑ com.netflix.discovery.EurekaClient：它实现了 Eureka Server 公开的所有 HTTP API 方法，有关这些方法的详细说明，请参见第 4.5 节“Eureka API”。
- ❑ org.springframework.cloud.client.discovery.DiscoveryClient：它是原生 Netflix EurekaClient 的 Spring Cloud 替代品。它提供了一个对所有发现客户端都有用的简单通用的 API。它有两种方法：getServices 和 getInstances。

```
private static final Logger LOGGER =
LoggerFactory.getLogger(ClientController.class);

@Autowired
private DiscoveryClient discoveryClient;

@GetMapping("/ping")
public List<ServiceInstance> ping() {
 List<ServiceInstance> instances =
discoveryClient.getInstances("CLIENT-SERVICE");
 LOGGER.info("INSTANCES: count={}", instances.size());
 instances.stream().forEach(it -> LOGGER.info("INSTANCE: id={},
port={}", it.getServiceId(), it.getPort()));
 return instances;
}
```

这里有一个与前面的实现有关的很有趣的事情。如果在服务启动后立即调用/ping 端点，则不会显示任何实例。这与响应缓存机制有关，在第 4.3.4 节中将对此有详细介绍。

## 4.3　高级配置设置

Eureka 的配置设置可分为以下 3 个部分。

- 服务器（Server）：它将自定义服务器行为。它包含前缀为 eureka.server.*的所有属性。完整的可用字段列表可以在 EurekaServerConfigBean 类中找到（https://github.com/spring-cloud/spring-cloud-netflix/blob/master/spring-cloud-netflix-eureka-server/src/main/java/org/springframework/cloud/netflix/eureka/server/EurekaServerConfigBean.java）。
- 客户端（Client）：这是 Eureka 客户端的两个可用属性部分中的第一个。它负责配置客户端如何查询注册表以查找其他服务。它包含前缀为 eureka.client.*的所有属性。有关可用字段的完整列表，可以参考 EurekaClientConfigBean 类（https://github.com/spring-cloud/spring-cloud-netflix/blob/master/spring-cloud-netflix-eureka-client/src/main/java/org/springframework/cloud/netflix/eureka/EurekaClientConfigBean.java）。
- 实例（Instance）：它将自定义 Eureka 客户端行为的当前实例，如端口或名称。它包含前缀为 eureka.instance.*的所有属性。有关可用字段的完整列表，可以参考 EurekaInstanceConfigBean 类（https://github.com/spring-cloud/spring-cloud-netflix/blob/master/spring-cloud-netflix-eureka-client/src/main/java/org/springframework/cloud/netflix/eureka/EurekaInstanceConfigBean.java）。

前文已经介绍了如何使用其中一些属性以获得所需的效果。本节后续内容将讨论与配置设置自定义相关的一些有趣场景。这里不需要解释所有属性，开发人员可以在之前列出的所有类的源代码所包含的注释中阅读到对它们的说明。

### 4.3.1　刷新注册表

现在可以回到上一个示例。虽然自我保护模式已被禁用，但服务器等待租约取消仍需要很长时间。这有几个原因，第一个原因是每个客户端服务每 30 秒（默认值）向服务器发送一次心跳，该间隔值可使用 eureka.instance.leaseRenewalIntervalInSeconds 属性进行配置。如果服务器没有收到心跳，那么它会在从注册表中删除实例之前等待 90 秒，然后才切断发送

到该实例的流量，这个等待的秒数值可以使用 eureka.instance.leaseExpirationDurationInSeconds 属性进行配置。这两个参数均在客户端设置。出于测试目的，我们定义一个很小的以秒为单位的值。

```
eureka:
 instance:
   leaseRenewalIntervalInSeconds: 1
   leaseExpirationDurationInSeconds: 2
```

还有一个应该在服务器端进行更改的属性。Eureka 在后台运行驱逐任务，负责检查是否仍在接收来自客户端的心跳。默认情况下，它每 60 秒触发一次。因此，即使租约续订的间隔和租约到期的持续时间被设置为相对较低的值，也仍然需要 60 秒（在最坏情况下）才能删除服务实例。可以使用 evictionIntervalTimerInMs 属性配置后续计时器在毫秒之间的延迟。请注意，该属性与前面所讨论的属性设置不同，它是以毫秒为单位的。

```
eureka:
 server:
   enableSelfPreservation: false
   evictionIntervalTimerInMs: 3000
```

所有必需参数都已在客户端和服务器端定义完毕。现在，开发人员可以再次运行发现服务器，然后使用-DPORT VM 参数在端口 8081、8082 和 8083 上运行客户端应用程序的 3 个实例。之后，再逐个关闭端口 8081 和 8082 上的实例，只需删除它们的进程即可。结果怎么样？被禁用的实例几乎立即就会从 Eureka 注册表中删除。

如图 4.4 所示就是来自 Eureka Server 的日志片段。

```
2017-11-02 21:44:56.533 INFO 40056 --- [a-EvictionTimer] c.n.e.registry.AbstractInstanceRegistry : Evicting 1 items (expired=1, evictionLimit=1)
2017-11-02 21:44:56.533 WARN 40056 --- [a-EvictionTimer] c.n.e.registry.AbstractInstanceRegistry : DS: Registry: expired lease for CLIENT-SERVICE/piomin:client-service:8082
2017-11-02 21:44:56.538 INFO 40056 --- [a-EvictionTimer] c.n.e.registry.AbstractInstanceRegistry : Cancelled instance CLIENT-SERVICE/piomin:client-service:8082 (replication=false)
2017-11-02 21:44:59.533 INFO 40056 --- [a-EvictionTimer] c.n.e.registry.AbstractInstanceRegistry : Running the evict task with compensationTime 0ms
2017-11-02 21:44:59.533 INFO 40056 --- [a-EvictionTimer] c.n.e.registry.AbstractInstanceRegistry : Evicting 1 items (expired=1, evictionLimit=1)
2017-11-02 21:44:59.533 WARN 40056 --- [a-EvictionTimer] c.n.e.registry.AbstractInstanceRegistry : DS: Registry: expired lease for CLIENT-SERVICE/piomin:client-service:8081
2017-11-02 21:44:59.534 INFO 40056 --- [a-EvictionTimer] c.n.e.registry.AbstractInstanceRegistry : Cancelled instance CLIENT-SERVICE/piomin:client-service:8081 (replication=false)
```

图 4.4　Eureka Server 的日志片段

在端口 8083 上仍有一个可用的实例正在运行。与自动保护模式相关的相应警告将打印在用户界面仪表板上。一些额外的信息（如租约到期状态或最后一分钟期间的续订数量）也可能很有趣。通过操纵所有这些属性，开发人员可以自定义过期租约删除进程的维护。但是，确保已定义的设置不会缺乏系统性能非常重要。还有一些其他元素对配置

的变化很敏感，如负载均衡器、网关和断路器等。如果禁用自我保护模式，Eureka 会打印一条警告消息，如图 4.5 所示。

| | |
|---|---|
| Lease expiration enabled | true |
| Renews threshold | 6 |
| Renews (last min) | 284 |

THE SELF PRESERVATION MODE IS TURNED OFF.THIS MAY NOT PROTECT INSTANCE EXPIRY IN CASE OF NETWORK/OTHER PROBLEMS.

DS Replicas

localhost

Instances currently registered with Eureka

| Application | AMIs | Availability Zones | Status |
|---|---|---|---|
| CLIENT-SERVICE | n/a (1) | (1) | UP (1) - piomin:client-service:8083 |

图 4.5　禁用自我保护模式之后显示的警告消息

## 4.3.2　更改实例标识符

在 Eureka 上注册的实例将按名称分组，但每个实例都必须发送一个唯一的 ID，服务器才能识别它。也许你已经注意到，instanceId 将显示在仪表板的每个服务组的 Status（状态）列中。Spring Cloud Eureka 会自动生成该数字，它等于以下字段的组合。

```
${spring.cloud.client.hostname}:${spring.application.name}:${spring.
application.instance_id:${server.port}}.
```

可以使用 eureka.instance.instanceId 属性轻松覆盖此标识符。出于测试目的，开发人员可以使用以下配置设置和-DSEQUENCE_NO = [n] VM 参数启动客户端应用程序的一些实例。其中，[n]是从 1 开始的顺序编号。以下是一个客户端应用程序的配置示例，该客户端应用程序将基于 SEQUENCE_NO 参数动态设置侦听端口和发现 instanceId。

```
server:
 port: 808${SEQUENCE_NO}
eureka:
 instance:
   instanceId: ${spring.application.name}-${SEQUENCE_NO}
```

可以在 Eureka 仪表板中查看其结果，如图 4.6 所示。

Instances currently registered with Eureka

| Application | AMIs | Availability Zones | Status |
|---|---|---|---|
| CLIENT-SERVICE | n/a (3) | (3) | UP (3) - client-service-2 , client-service-3 , client-service-1 |

图 4.6　动态设置的实例标识符

### 4.3.3 选择使用 IP 地址

默认情况下，所有实例都在其主机名下注册。假设开发人员在网络上启用了域名系统（Domain Name System，DNS），这是一种非常方便的方法。但是，对于用作企业中的微服务环境的一组服务器来说，DNS 并不常见。笔者就是这种情况。除了将主机名及其 IP 地址添加到所有 Linux 机器上的/etc/hosts 文件之外，没有其他办法。此解决方案的替代方法是更改注册过程配置设置以通告服务的 IP 地址而不是主机名。要实现此目的，应在客户端将 eureka.instance.preferIpAddress 属性设置为 true。执行该项设置之后，虽然注册表中的每个服务实例在 Eureka 仪表板中仍将显示包含主机名的 instanceId，但如果单击此链接，则将根据 IP 地址执行重定向。负责通过 HTTP 调用其他服务的 Ribbon 客户端也遵循相同的原则。

如果开发人员决定使用 IP 地址作为确定服务的网络位置的主要方法，则可能会遇到问题。简而言之，如果开发人员为自己的计算机分配了多个网络接口，则可能会出现此问题。例如，在笔者工作过的一个企业中，其管理模式（从个人工作站到服务器的连接）和生产模式（两台服务器之间的连接）具有不同的网络。因此，每台服务器都有两个分配有不同 IP 前缀的网络接口。要选择正确的接口，可以在 application.yml 配置文件中定义被忽略的模式列表。例如，假设要忽略名称以 eth1 开头的所有接口，则可以进行以下定义。

```
spring:
  cloud:
    inetutils:
      ignoredInterfaces:
        - eth1*
```

还有另一种方法也可以达到这种效果。开发人员可以定义应该首选的网络地址：

```
spring:
  cloud:
    inetutils:
      preferredNetworks:
        - 192.168
```

### 4.3.4 响应缓存

Eureka Server 默认会缓存响应。缓存每 30 秒失效一次。可以通过调用 HTTP API 端

点/eureka/apps 轻松检查它。如果开发人员在注册客户端应用程序后立即调用它，则将发现它仍未在响应中返回。30 秒后再试一次，即可看到新实例出现。

可以使用 responseCacheUpdateIntervalMs 属性覆盖响应缓存超时。有趣的是，使用 Eureka 仪表板显示已注册实例列表时却没有缓存。因为它与 REST API 相反，它绕过了响应缓存。

```
eureka:
 server:
   responseCacheUpdateIntervalMs: 3000
```

开发人员应该记住，Eureka 注册表也会在客户端缓存。因此，即使我们更改了服务器上的缓存超时，它仍可能需要一些时间才能被客户端刷新。默认情况下，注册表会在每 30 秒调度一次的异步后台任务中定期刷新。我们可以通过声明 registryFetchInterval Seconds 属性来覆盖此设置。与最近一次获取尝试相比，它仅获取增量变化。可以使用 shouldDisableDelta 属性禁用此选项。我们已经在服务器端和客户端都定义了 3 秒的超时，此时如果使用/eureka/apps 设置启动示例应用程序，则在首次尝试时将可能显示新注册的服务实例。除非在客户端的缓存很有意义，否则我们并不确定服务器端缓存的意义，特别是考虑到 Eureka 没有任何后端存储的情况下。就个人而言，笔者从来没有对这些属性的值进行过任何修改，但笔者猜测在某些情况下它也可能很重要。例如，开发人员使用 Eureka 开发单元测试并且需要立即响应而不进行缓存。

```
eureka:
 client:
   registryFetchIntervalSeconds: 3
   shouldDisableDelta: true
```

## 4.4　启用客户端和服务器之间的安全通信

到目前为止，Eureka Server 没有对任何客户端的连接进行过身份验证。在开发模式中，安全性并不像生产模式那么重要。缺乏它可能是一个问题。我们希望通过基本身份验证保护发现服务器，以防止未经授权访问任何知道其网络地址的服务。尽管 Spring Cloud 参考资料声称 HTTP 基本身份验证将自动添加到 Eureka 客户端，但开发人员仍必须在附属项目中包含一个具有安全性的启动器。

```
<dependency>
    <groupId>org.springframework.boot</groupId>
```

```
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

然后，开发人员应该通过更改 application.yml 文件中的配置设置来启用安全性，并且设置默认凭据。

```
security:
 basic:
   enabled: true
 user:
   name: admin
   password: admin123
```

现在，所有 HTTP API 端点和 Eureka 仪表板都是安全的。要在客户端启用基本身份验证模式，应在 URL 连接地址中提供凭据，如以下配置设置中所示。实现安全发现的示例应用程序可在同一存储库（https://github.com/piomin/sample-spring-cloud-netflix.git）中找到，并且可用作基本示例，但开发人员需要切换到 security 分支（https://github.com/piomin/sample-spring-cloud-netflix/tree/security）。以下是在客户端启用 HTTP 基本身份验证的配置。

```
eureka:
 client:
   serviceUrl:
     defaultZone: http://admin:admin123@localhost:8761/eureka/
```

对于更高级的使用，如在发现客户端和服务器之间使用证书身份验证方式的安全 SSL 连接，开发人员应该提供 DiscoveryClientOptionalArgs 的自定义实现。在本书第 12 章“保护 API 的安全”中将会讨论这样一个示例，该示例将致力于保护 Spring Cloud 应用程序的安全。

保护服务器端的安全性是一回事，注册安全应用程序则是另一回事。现在让我们来看一看如何注册安全服务。

（1）要为 Spring Boot 应用程序启用安全套接层（Secure Sockets Layer，SSL），需要从生成自签名证书（Self-Signed Certificate）开始。建议使用 keytool，它可以在 bin 目录中的 JRE 根目录下使用。

```
keytool -genkey -alias client -storetype PKCS12 -keyalg RSA -
keysize 2048 -keystore keystore.p12 -validity 3650
```

（2）输入所需数据并将生成的密钥库文件 keystore.p12 复制到应用程序的 src/main/resources 目录中，而接下来的步骤则是使用 application.yml 中的配置属性为 Spring Boot

启用 HTTPS。

```
server:
 port: ${PORT:8081}
 ssl:
   key-store: classpath:keystore.p12
   key-store-password: 123456
   keyStoreType: PKCS12
   keyAlias: client
```

（3）在运行应用程序之后，开发人员应该能够调用安全端点 https://localhost:8761/info。此时还需要在 Eureka 客户端实例配置中执行一些更改。

```
eureka:
 instance:
   securePortEnabled: true
   nonSecurePortEnabled: false
   statusPageUrl: https://${eureka.hostname}:${server.port}/info
   healthCheckUrl: https://${eureka.hostname}:${server.port}/health
   homePageUrl: https://${eureka.hostname}:${server.port}/
```

## 4.5　Eureka API

Spring Cloud Netflix 提供了一个用 Java 编写的客户端，它隐藏了开发人员的 Eureka HTTP API。如果开发人员使用除 Spring 之外的其他框架，则 Netflix OSS 提供了一个可以作为依赖项包含的普通 Eureka 客户端。但是，开发人员很容易就能想到一些需要直接调用 Eureka API 的情况。例如，如果应用程序是使用除 Java 之外的其他语言编写的，或者开发人员在持续交付（Continuous Delivery）过程中需要已注册服务列表之类的信息，这些都需要直接调用 Eureka API。表 4.1 提供了 Eureka API 的快速参考。

表 4.1　Eureka API 快速参考

| HTTP 端点 | 说　　明 |
|---|---|
| POST /eureka/apps/appID | 将新服务实例添加到注册表 |
| DELETE /eureka/apps/appID/instanceID | 从注册表中删除服务实例 |
| PUT /eureka/apps/appID/instanceID | 向服务器发送心跳 |
| GET /eureka/apps | 获取有关所有已注册服务实例列表的详细信息 |

续表

| HTTP 端点 | 说　明 |
| --- | --- |
| GET /eureka/apps/appID | 获取有关特定服务的所有已注册实例列表的详细信息 |
| GET /eureka/apps/appID/instanceID | 获取有关单个服务实例的详细信息 |
| PUT /eureka/apps/appID/instanceID/metadata?key=value | 更新元数据参数 |
| GET /eureka/instances/instanceID | 获取有关具有特定 ID 的所有已注册实例的详细信息 |
| PUT /eureka/apps/appID/instanceID/status？value=DOWN | 更新实例的状态 |

## 4.6　副本和高可用性

前文已经讨论了一些有用的 Eureka 设置，但是到目前为止，我们只分析了一个具有单台服务发现服务器的系统。这样的配置虽然是有效的，但仅限于开发模式。对于生产模式来说，至少需要运行两台发现服务器，以防其中一台发生故障或发生网络问题。根据定义，Eureka 是为可用性（Availability）和弹性（Resiliency）而构建的，这是 Netflix 开发的两个主要支柱，但它不提供标准的集群机制，如领导选举或自动加入集群。它基于对等复制模型。这意味着所有服务器都复制数据并将心跳发送到所有对等体，这些对等体在当前服务器节点的配置中设置。这种算法对于包含数据简单而有效，但它也有一些缺点。它限制了可伸缩性，因为每个节点都必须承受服务器上的整个写入负载。

### 4.6.1　样本解决方案的架构

有趣的是，副本（Replication）机制是开发人员开始使用新版 Eureka Server 的主要动机之一。Eureka 2.0 目前仍在积极开发中。除了优化的副本机制之外，它还将提供一些有趣的功能，如从服务器到客户端的推送模型（用于推送注册列表中的任何更改）、自动扩展的服务器和丰富的仪表板等。这个解决方案似乎很有实践意义，但 Spring Cloud Netflix 使用的仍然是版本 1。说实话，我们尚未发现任何迁移到版本 2 的计划，目前 Dalure.SR4 版本列车的 Eureka 版本是 1.6.2。服务器端的集群机制的配置可以归结为一件事，即使用 eureka.client.* 属性部分设置另一个发现服务器的 URL。所选定的服务器将只会在其他服务器中注册自己，而这些服务器将被选为已创建集群的一部分。要显示该解决方案在实践中的工作原理，最佳方式当然是通过示例。

现在就让我们从示例系统的架构开始。如图 4.7 所示，我们所有的应用程序都将在不同的端口上以本地方式运行。在此阶段，我们必须介绍基于 Netflix Zuul 的 API 网关示例，这对于进行负载均衡测试是有帮助的。本示例将测试在不同区域（Zone）中注册的服务的 3 个实例之间的负载均衡。

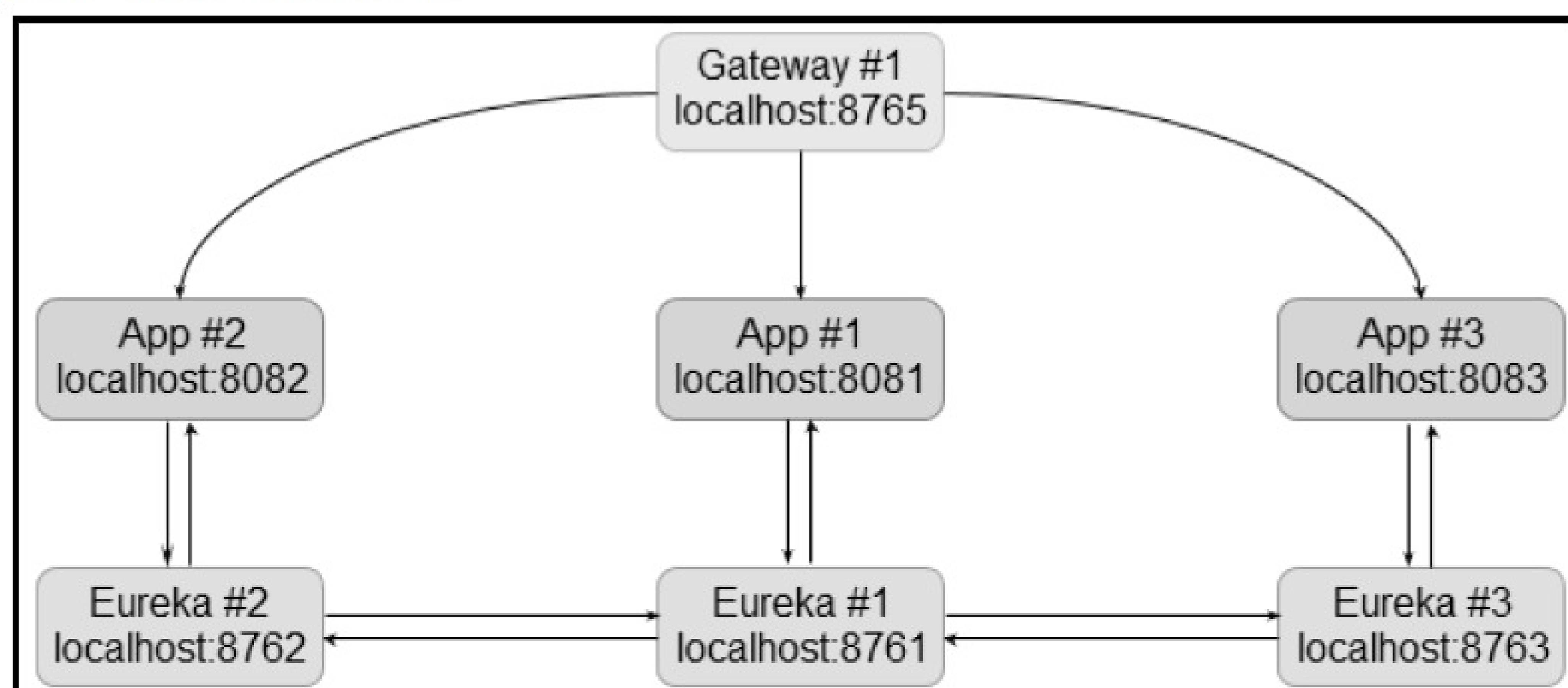



图 4.7　示例系统架构

### 4.6.2　构建示例应用程序

对于 Eureka Server 来说，可以在配置属性中定义所有必需的更改。在 application.yml 文件中，我们为发现服务的每个实例定义了 3 个不同的配置文件。现在，如果要尝试运行嵌入在 Spring Boot 应用程序中的 Eureka Server，则需要通过提供 VM 参数-Dspring.profiles.active=peer[n]来激活特定的配置文件，其中，[n]是实例的顺序编号。

```
spring:
 profiles: peer1
eureka:
 instance:
   hostname: peer1
   metadataMap:
     zone: zone1
 client:
   serviceUrl:
     defaultZone:
http://localhost:8762/eureka/,http://localhost:8763/eureka/
server:
 port: ${PORT:8761}
```

```
---
spring:
  profiles: peer2
eureka:
 instance:
   hostname: peer2
   metadataMap:
     zone: zone2
 client:
   serviceUrl:
     defaultZone:
http://localhost:8761/eureka/,http://localhost:8763/eureka/
server:
 port: ${PORT:8762}

---
spring:
  profiles: peer3
eureka:
  instance:
   hostname: peer3
   metadataMap:
     zone: zone3
 client:
   serviceUrl:
     defaultZone:
http://localhost:8761/eureka/,http://localhost:8762/eureka/
server:
 port: ${PORT:8763}
```

在使用不同的配置文件名运行所有 3 个 Eureka 实例后，我们就已经创建了一个本地发现集群。如果在启动后查看任何实例的 Eureka 仪表板，那么它看起来始终是一样的，我们可以看到 3 个 DISCOVERY-SERVICE 实例，如图 4.8 所示。

DS Replicas

localhost

Instances currently registered with Eureka

| Application | AMIs | Availability Zones | Status |
| --- | --- | --- | --- |
| DISCOVERY-SERVICE | n/a (3) | (3) | UP (3) - minkowp-l.p4.org:discovery-service:8763 , minkowp-l.p4.org:discovery-service:8762 , minkowp-l.p4.org:discovery-service:8761 |

图 4.8　仪表板上的 3 个 DISCOVERY-SERVICE 实例

下一步是运行客户端应用程序。项目中的配置设置与使用 Eureka Server 的应用程序

的配置设置非常相似。defaultZone 字段中提供的地址顺序决定了对不同发现服务的连接尝试的顺序。如果无法建立与第一个服务器的连接，那么它将尝试从列表中连接第二个服务器，以此类推。与前文所述一样，开发人员应该设置 VM 参数-Dspring.profiles.active = zone[n]来选择正确的配置文件。这里还建议设置-Xmx192m 参数。请记住，我们要在本地测试所有服务，如果没有为 Spring Cloud 应用程序设置任何内存限制，则启动后消耗大约 350MB 的堆，并且总的内存使用大约为 600MB。除非计算机上有大量内存，否则很难在本地机器上运行多个微服务实例。

```
spring:
 profiles: zone1
eureka:
 client:
   serviceUrl:
     defaultZone:
http://localhost:8761/eureka/,http://localhost:8762/eureka/,http://localhost:
8763/eureka/
server:
  port: ${PORT:8081}

---
spring:
  profiles: zone2
eureka:
  client:
   serviceUrl:
     defaultZone:
http://localhost:8762/eureka/,http://localhost:8761/eureka/,http://localhost:
8763/eureka/
server:
  port: ${PORT:8082}

---
spring:
  profiles: zone3
eureka:
  client:
   serviceUrl:
     defaultZone:
http://localhost:8763/eureka/,http://localhost:8761/eureka/,http://localhost:
8762/eureka/
```

```
server:
 port: ${PORT:8083}
```

现在再来看一下 Eureka 仪表板。尽管应用程序最初只连接到发现服务的一个实例，但我们在任何地方都注册了 3 个客户端服务实例。无论进入哪个发现服务实例的仪表板，结果都是相同的。这就是该练习的确切目的。现在可以创建一些额外的实现以证明一切都已如预期一样工作，如图 4.9 所示。

DS Replicas

localhost

Instances currently registered with Eureka

| Application | AMIs | Availability Zones | Status |
| --- | --- | --- | --- |
| CLIENT-SERVICE | n/a (3) | (3) | UP (3) - minkowp-l.p4.org:client-service:9110 , minkowp-l.p4.org:client-service:9100 , minkowp-l.p4.org:client-service:9120 |
| DISCOVERY-SERVICE | n/a (3) | (3) | UP (3) - minkowp-l.p4.org:discovery-service:8763 , minkowp-l.p4.org:discovery-service:8762 , minkowp-l.p4.org:discovery-service:8761 |

图 4.9　Eureka 仪表板

客户端应用程序除了将打印所选配置文件名称的 REST 端点公开之外并无其他操作。配置文件名称指向特定应用程序实例的主发现服务实例。以下@RestController 实现非常简单，它将打印当前区域的名称。

```
@RestController
public class ClientController {
  @Value("${spring.profiles}")
  private String zone;

  @GetMapping("/ping")
  public String ping() {
    return "I'm in zone " + zone;
  }

}
```

最后，开发人员可以继续实现 API 网关。当然，详细介绍 Zuul、Netflix 的 API 网关和路由器提供的功能超出了本章的范围（后续章节将对此展开详细的讨论）。Zuul 现在将有助于测试我们的示例解决方案，因为它能够检索在发现服务器中注册的服务列表，并在客户端应用程序的所有正在运行的实例之间执行负载均衡。正如以下配置片段所示，开发人员可以使用侦听端口 8763 的发现服务器。所有包含/api/client/**路径的传入请求都将路由到 client-service。

```
zuul:
 prefix: /api
```

```
 routes:
  client:
   path: /client/**
   serviceId: client-service

eureka:
 client:
  serviceUrl:
   defaultZone: http://localhost:8763/eureka/
  registerWithEureka: false
```

现在来继续进行测试。开发人员可以使用 java -jar 命令启动 Zuul 代理（Proxy）的应用程序，与以前的服务不同，这次不需要设置任何其他参数，包括配置文件名称。默认情况下，它将与编号#3 的发现服务连接。要通过 Zuul 代理调用客户端 API，开发人员必须在 Web 浏览器中输入以下地址 http://localhost:8765/api/client/ping。其结果如图 4.10 所示。

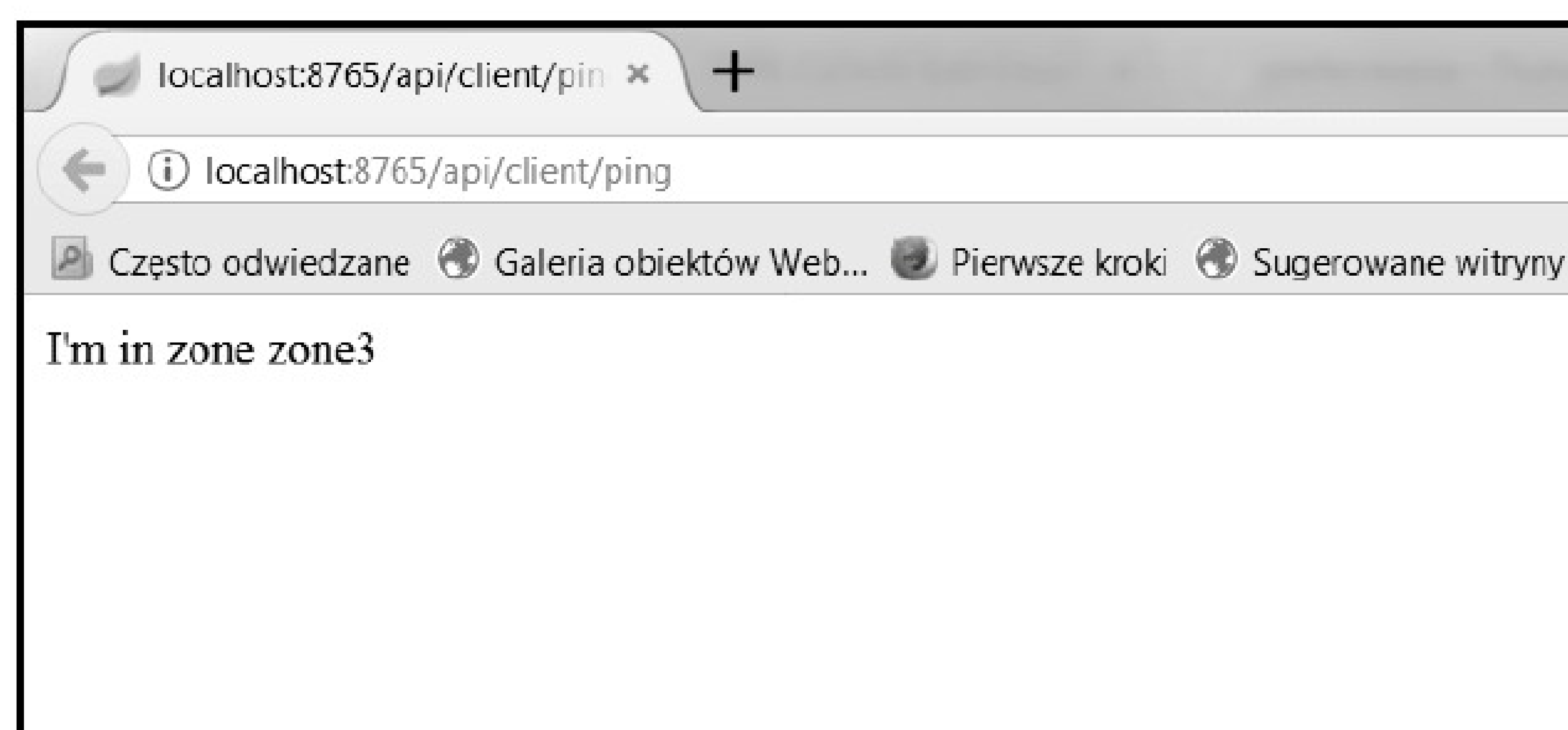



图 4.10　连接到编号#3 的发现服务

如果连续多次重试请求，则应按 1:1:1 的比例在所有现有 client-service 实例之间进行负载均衡，尽管我们的网关仅连接到编号#3 的发现服务。此示例完整演示了如何使用多个 Eureka 实例构建服务发现。

上述示例应用程序可在 GitHub（https://github.com/piomin/sample-spring-cloud-netflix.git）上的 cluster 分支（https://github.com/piomin/sample-spring-cloud-Netflix/tree/cluster_no_zones）中获得。

### 4.6.3　故障转移

也许有读者会问：如果一个服务发现实例发生故障会怎么样呢？为了检查集群在发

生故障时的行为方式，不妨稍微修改一下之前的示例。现在，Zuul 具有一个故障转移（Failover）连接，它可以连接到其配置设置中设置的端口 8762 上可用的第二个服务发现。出于测试目的，可以关闭端口 8763 上可用的第三个发现服务实例。

```
eureka:
 client:
   serviceUrl:
     defaultZone:
http://localhost:8763/eureka/,http://localhost:8762/eureka/
   registerWithEureka: false
```

目前的情况如图 4.11 所示。通过调用 http://localhost:8765/api/client/ping 地址下可用的网关端点，可以按与先前相同的方式执行测试。其结果也与之前的测试相同，负载均衡在所有 3 个客户端服务实例中按预期平均执行。虽然已禁用发现服务＃3，但是其他两个实例仍然能够相互通信，并且只要它是活动的，就可以获得有关从实例#3 复制的第 3 个客户端应用程序实例的网络位置的信息。现在，即使重新启动网关，它仍然能够按顺序使用第二个地址连接发现集群（这是在 http://localhost:8762/eureka 的 defaultZone 字段中设置的）。这同样适用于客户端应用程序的第 3 个实例，而第 3 个实例又会将发现服务#1 作为备份连接。

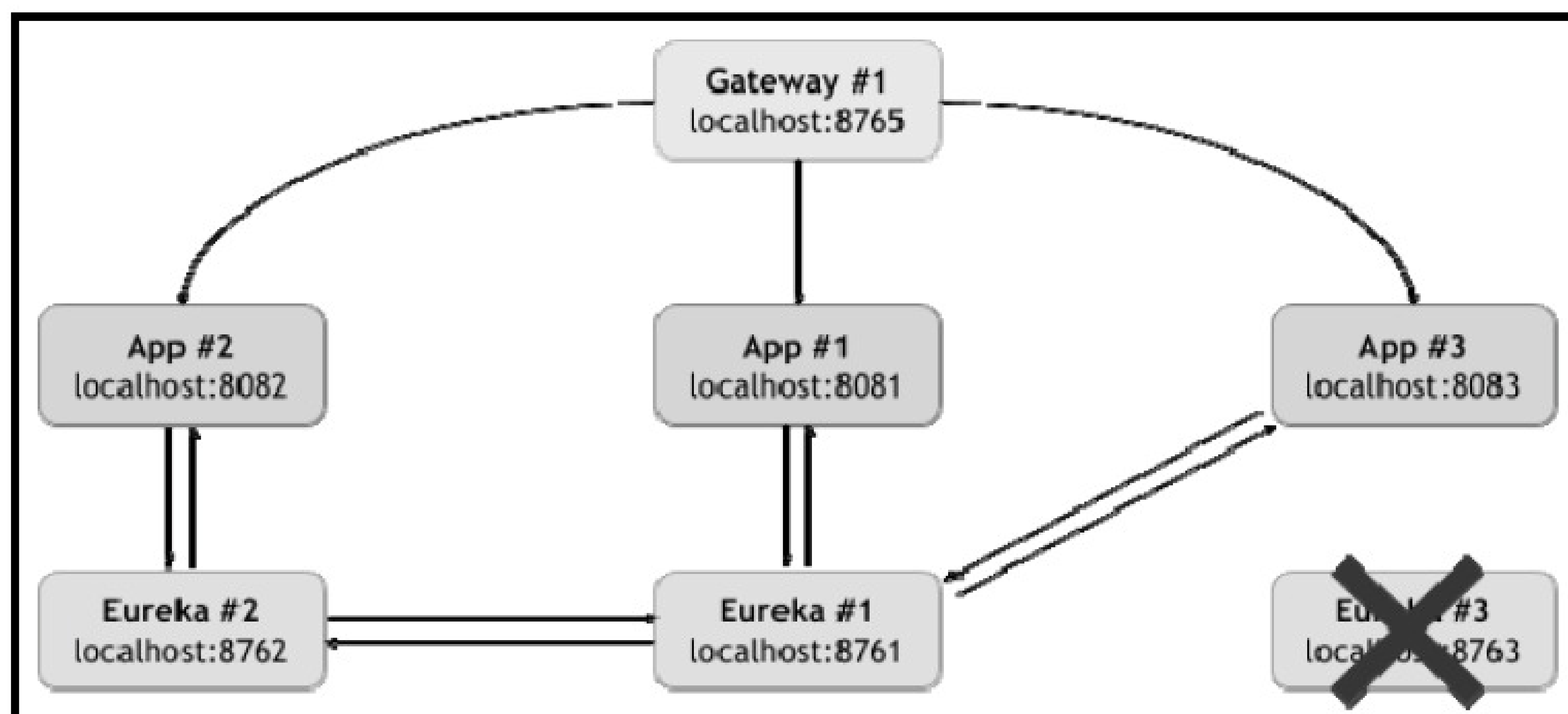



图 4.11　故障转移机制

## 4.7　区　　域

在大多数情况下，基于集群（Cluster）的对等复制模型（Peer-to-Peer Replication Model）是一种很好的方式，但并不总是够用。Eureka 还有一个更有趣的功能，在集群环境中非

常有用。事实上，区域机制（Zone Mechanism）是其默认行为。即使开发人员有一个独立的服务发现实例，每个客户端的属性也必须在配置设置中将其设置为 eureka.client.serviceUrl.defaultZone。这在什么时候对我们很有用呢？为了分析清楚，不妨回到第 4.6 节的示例。假设现在我们的环境分为 3 个不同的物理网络，或者只有 3 台不同的机器处理传入的请求。当然，发现服务仍然在逻辑上分组在集群中，但每个实例都放在一个单独的区域中。每个客户端应用程序都将在与其主发现服务器相同的区域中注册。我们将启动 3 个实例（而不是 Zuul 网关的一个实例），每个实例用于一个区域。如果请求进入网关，那么它所选择的客户端应该在尝试调用另一个区域中注册的服务之前，优先利用同一区域内的服务。如图 4.12 所示是当前系统架构的可视化示意图。当然，出于示例目的，该架构被简化为能够在单台本地机器上运行。在现实世界中，正如前文所述，它将在 3 台不同的机器上运行，甚至在 3 组不同的机器上启动，从物理上分隔为其他网络。

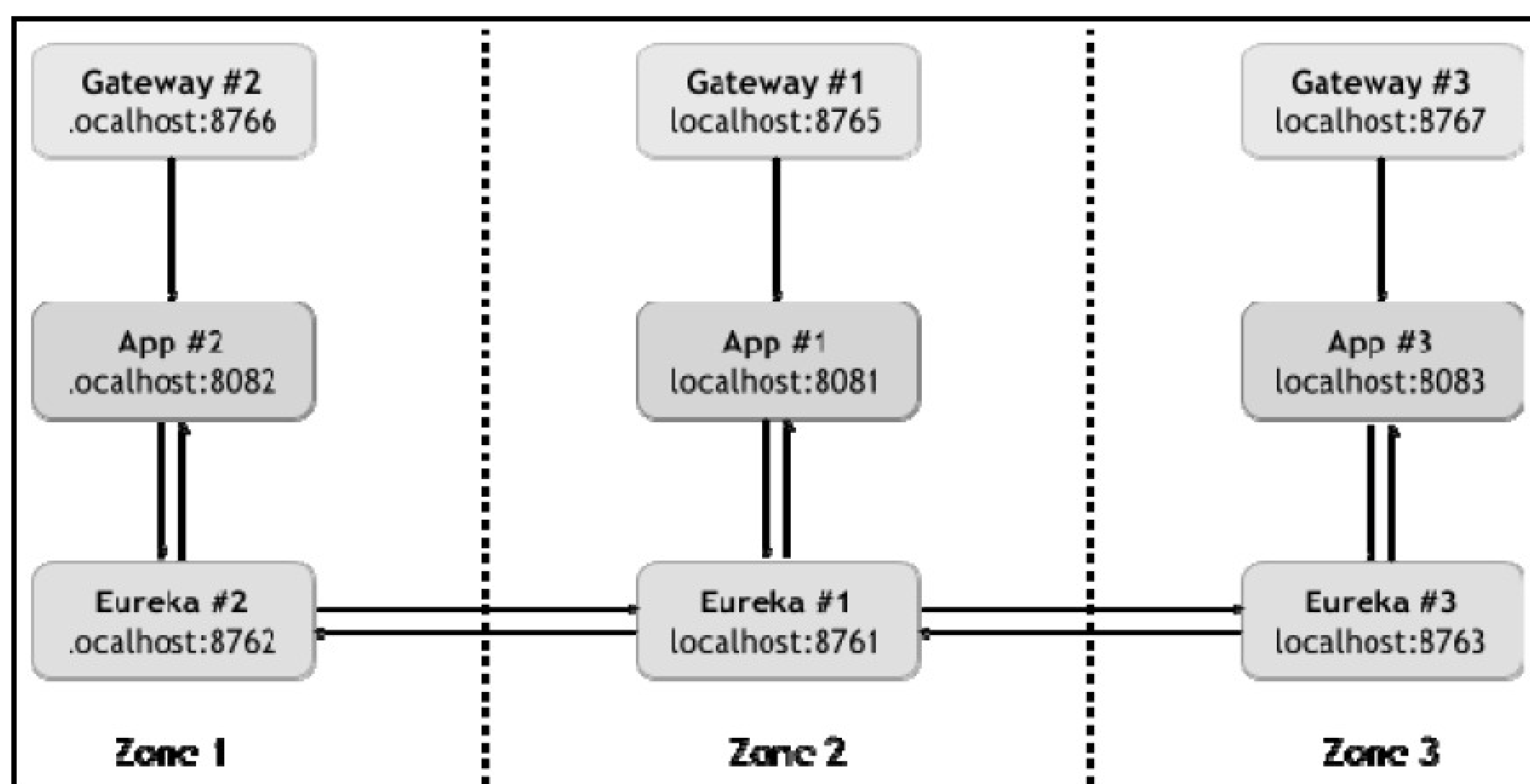



图 4.12　区域机制简化示意

## 4.7.1　具有独立服务器的区域

在本阶段，我们应该强调一件重要的事情，即分区机制只在客户端实现。这意味着服务发现实例未指定给任何区域。因此，图 4.12 可能会让开发人员略微产生一些混淆，但它指出了哪个 Eureka 是在特定区域中注册的所有客户端应用程序和网关的默认服务发现。我们的目的是检查高可用性模式中的机制，但我们也可以仅使用单个发现服务器来构建它。图 4.13 演示了与图 4.12 类似的情况，但它假定所有应用程序只存在一个发现服务器。

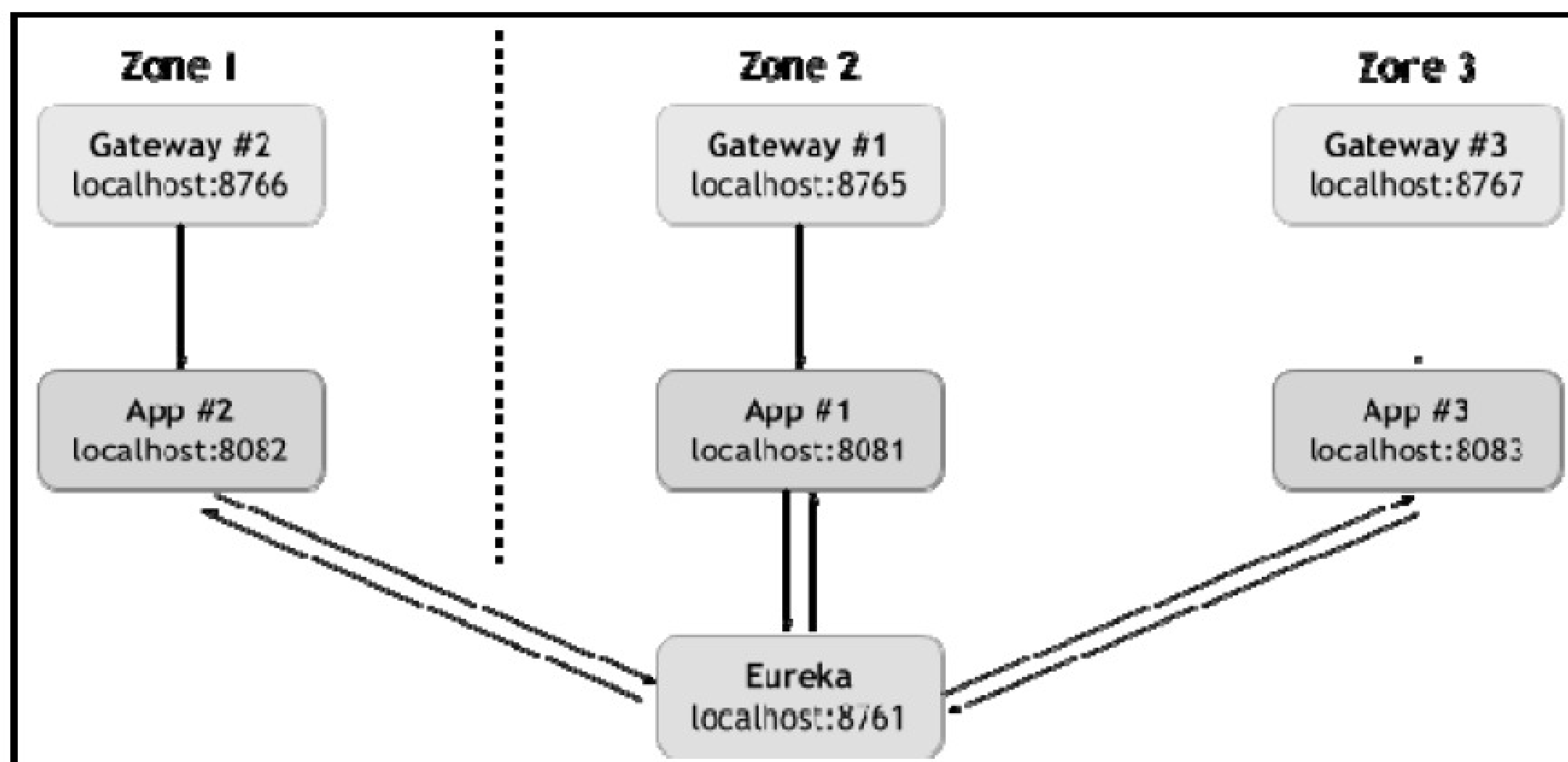



图 4.13　具有独立服务器的区域

## 4.7.2　构建示例应用程序

要启用区域处理机制，开发人员需要在客户端和网关的配置设置中执行一些更改。以下是客户端应用程序中修改后的 application.yml 文件。

```
spring:
 profiles: zone1
eureka:
 instance:
   metadataMap:
     zone: zone1
 client:
   serviceUrl:
     defaultZone:
http://localhost:8761/eureka/,http://localhost:8762/eureka/,http://localhost:
8763/eureka/
```

唯一需要更新的是 eureka.instance.metadataMap.zone 属性，在该属性中可以设置区域的名称和已经注册的服务。

在网关配置中还必须进行更多更改。首先，需要添加 3 个配置文件，以便能够运行在 3 个不同的区域和 3 个不同的发现服务器中注册的应用程序。现在，在启动网关应用程序时，开发人员应该设置 VM 参数-Dspring.profiles.active = zone[n]以选择正确的配置文件。

与 client-service 类似，开发人员还必须在配置设置中添加 eureka.instance.metadataMap.zone 属性。还有一个属性 eureka.client.preferSameZoneEureka 也值得一提，这是在示例中第一次使用的，如果网关应该优先选择在同一区域中注册的客户端应用程序的实例，则该属性必须等于 true。

```
spring:
 profiles: zone1
eureka:
 client:
   serviceUrl:
     defaultZone: http://localhost:8761/eureka/
     registerWithEureka: false
     preferSameZoneEureka: true
 instance:
   metadataMap:
     zone: zone1
server:
 port: ${PORT:8765}

---
spring:
 profiles: zone2
eureka:
 client:
   serviceUrl:
     defaultZone: http://localhost:8762/eureka/
     registerWithEureka: false
     preferSameZoneEureka: true
  instance:
   metadataMap:
     zone: zone2
server:
 port: ${PORT:8766}

---
spring:
 profiles: zone3
eureka:
 client:
   serviceUrl:
     defaultZone: http://localhost:8763/eureka/
     registerWithEureka: false
```

```
      preferSameZoneEureka: true
  instance:
    metadataMap:
      zone: zone3
server:
  port: ${PORT:8767}
```

在启动发现、客户端和网关应用程序的所有实例后，开发人员可以尝试调用在 http://localhost:8765/api/client/ping、http://localhost:8766/api/client/ping 和 http://localhost:8767/api/client/ping 地址下可用的端点。它们中的每一个都将始终与在同一区域中注册的客户端实例进行通信。因此，与没有首选区域的测试不同，端口 8765 下可用的第一个网关实例将始终在调用 ping 端点时显示 I'm in zone zone1，如图 4.14 所示。

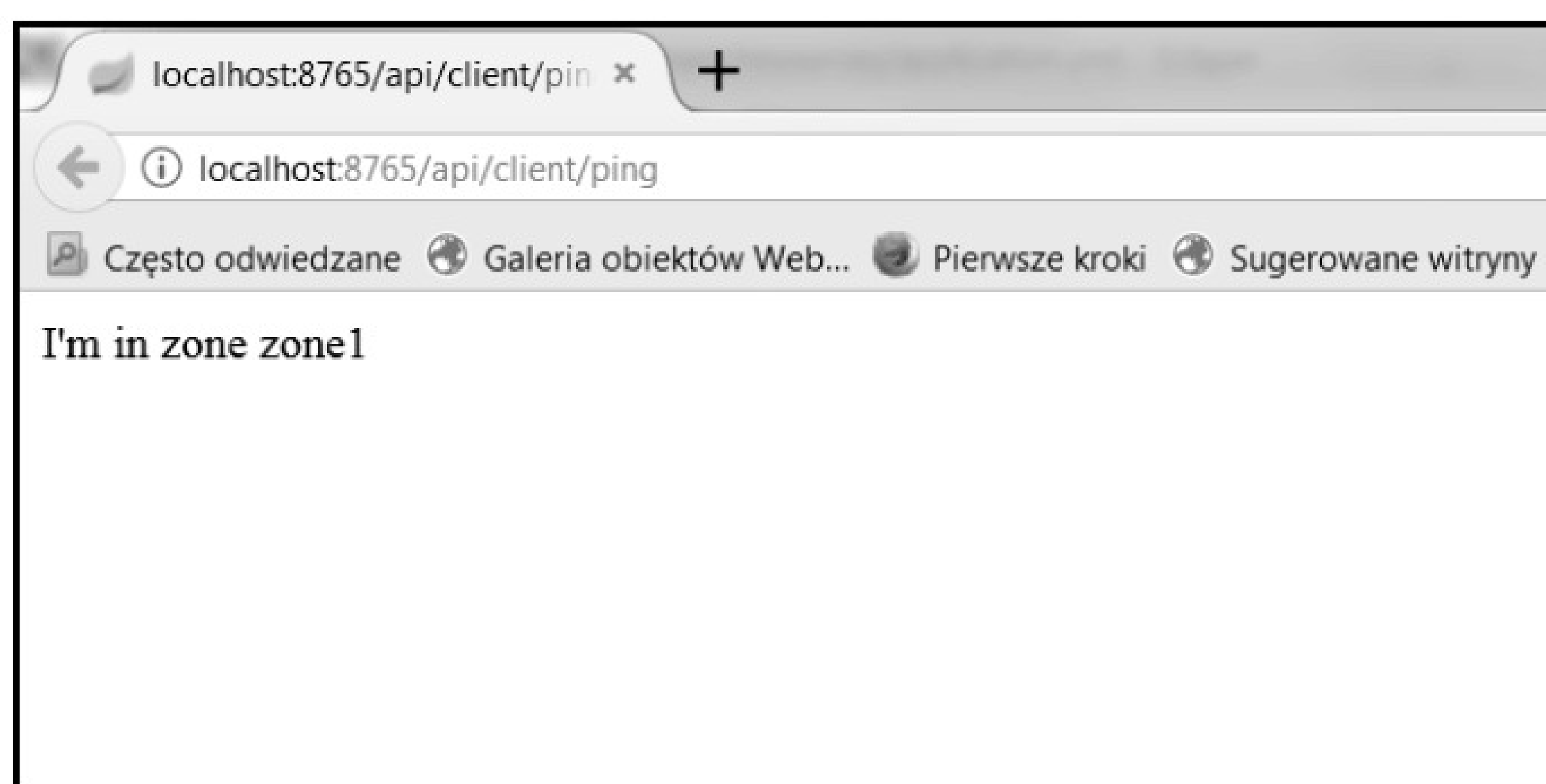



图 4.14　区域处理机制将始终选择相同区域的客户端实例

当客户端编号#1 不可用时会发生什么呢？传入请求将在客户端应用程序的两个其他实例之间进行 50/50 的负载均衡，因为它们都位于与网关#1 不同的区域中。

## 4.8　小　　结

本章首次使用了 Spring Cloud 开发应用程序。在笔者看来，学习使用微服务框架的最佳方法是尝试了解如何正确实现服务发现。本章从最简单的用例和示例开始，全面介绍了 Netflix OSS Eureka 项目提供的高级功能和可直接应用于生产模式的功能。本章展示了如何在 5 分钟内创建和运行基本客户端和独立发现服务器。基于该实现，本章还介绍了如何自定义 Eureka 客户端和服务器以满足开发人员的特定需求，并且重点讨论了网络或应用程序故障等负面情况的应对。本章还详细讨论了 REST API 或用户界面仪表板等功

能。最后，本章还演示了如何使用 Eureka 的机制（如副本、区域和高可用性等）创建可用于生产模式的环境。有了这些知识，开发人员就应该能够正确选择 Eureka 的功能，并通过这些功能构建适应基于微服务架构要求的服务发现。

在掌握了服务发现之后，开发人员即可进入基于微服务的体系结构中的下一个基本元素，即配置服务器。发现和配置服务通常都基于键/值存储，因此它们可以使用相同的产品提供。但是，由于 Eureka 仅专注于发现，所以 Spring Cloud 引入了自己的框架 Spring Cloud Config 来管理分布式配置。

# 第 5 章　使用 Spring Cloud Config 进行分布式配置

现在是在我们的架构中引入新元素的最佳时机，这个新元素就是分布式配置服务器（Distributed Configuration Server）。与服务发现类似，这是围绕微服务的关键概念之一。在本书第 4 章中，已经详细讨论了如何在服务器端和客户端上准备发现。但是到目前为止，我们始终是使用放置在胖 JAR 文件中的属性为应用程序提供配置。这种方法有一个很大的缺点，那就是它需要重新编译和重新部署微服务的实例。Spring Boot 支持另一种方法，它假定使用存储在胖 JAR 之外的文件系统中的显式配置。使用 spring.config.location 属性可以在启动期间为应用程序轻松配置它。这种方法不需要重新部署，但它也不是没有缺点。在使用大量微服务的情况下，基于放置在文件系统中的显式文件的配置管理可能真的很麻烦。另外，如果每个微服务都有很多实例，而每个实例又都有一个特定的配置。那么，采用这种方法的麻烦程度简直不敢想象。

无论如何，分布式配置是云原生环境中非常流行的标准。Spring Cloud Config 可以为分布式系统中的外部化配置提供服务器端和客户端支持。通过该解决方案，开发人员可以在一个中心位置管理所有环境中的应用程序的外部属性。这个概念非常简单，而且易于实现。服务器只是公开 HTTP 和基于资源的 API 接口，它们将以 JSON、YAML 或 properties 格式返回 property 文件。此外，它还会对返回的属性值执行解密和加密操作。客户端需要从服务器获取配置设置，并且如果在服务器端启用了加密之类的功能，则客户端还需要对它们进行解密处理。

配置数据可以存储在不同的存储库（Repository）中。EnvironmentRepository 的默认实现将使用 Git 后端（Backend）。也可以设置其他版本控制系统（Version Control System，VCS），如 SVN。如果开发人员不想利用 VCS 系统提供的功能作为后端，则可以使用文件系统或 Vault。Vault 是一种用于管理机密的工具，它可以用于存储和控制对令牌、密码、证书和 API 密钥等资源的访问。

本章将要讨论的主题包括：

- 由 Spring Cloud Config Server 公开的 HTTP API。
- 服务器端的不同类型的存储库后端。
- 与服务发现集成。

- ❑ 使用 Spring Cloud Bus 和消息代理自动重新加载配置。

## 5.1　HTTP API 资源简介

Config Server 将提供 HTTP API，它可以通过各种方式调用。以下端点都是可用的。

- ❑ /{application}/{profile}[/{label}]：以 JSON 格式返回数据；label 参数是可选的。
- ❑ /{application}-{profile}.yml：返回 YAML 格式。
- ❑ /{label}/{application}-{profile}.yml：这是前一个端点的变体，开发人员可以传递一个可选的 label 参数。
- ❑ /{application}-{profile}.properties：返回 properties 文件使用的简单键/值格式。
- ❑ /{label}/{application}-{profile}.properties：这是前一个端点的变体，开发人员可以传递一个可选的 label 参数。

从客户端的角度来看，application 参数是应用程序的名称，它取自 spring.application.name 或 spring.config.name 的属性，profile 是活动配置文件或以逗号分隔的活动配置文件列表。最后一个可用的参数 label 是一个可选属性，只有在使用 Git 作为后端存储时才很重要。它将设置 Git 分支的名称以进行配置，默认为 master。

现在可以从最简单的基于文件系统后端的例子开始。默认情况下，Spring Cloud Config Server 将尝试从 Git 存储库获取配置数据。要启用原生配置文件，开发人员应该将 spring.profiles.active 选项设置为 native 来启动服务器。它将搜索存储在以下位置的文件：classpath:/、classpath:/config、file:./、file:./config。这意味着 properties 或 YAML 文件也可以放在 JAR 文件中。为测试起见，我们已经在 src/main/resources 中创建了一个 config 文件夹。我们的配置文件将存储在该位置中。现在需要回到本书第 4 章的示例中。在第 4 章中已经介绍了集群发现环境的配置，该环境中的每个客户端服务实例都在不同的区域中启动。它有 3 个可用区域和 3 个客户端实例，每个实例都在 application.yml 文件中有自己的配置文件。该示例的源代码在 config 分支中可用，以下是其链接。

```
https://github.com/piomin/sample-spring-cloud-netflix/tree/config
```

我们当前的任务是将该配置迁移到 Spring Cloud Config Server。这里不妨自我提醒一下这个示例的属性。以下是用于客户端应用程序的第一个实例的配置文件设置。根据所选的配置文件，有一个修改实例运行的端口、默认发现服务器 URL 和区域名称。

```
---
spring:
```

```
 profiles: zone1

eureka:
 instance:
  metadataMap:
    zone: zone1
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/

server:
 port: ${PORT:8081}
```

在前面所介绍的示例中，为简单起见，已经将所有配置文件设置放在单个 application.yml 文件中。该文件也可以分为 3 个不同的文件，其名称包括配置文件 application-zone1.yml、application-zone2.yml 和 application-zone3.yml。当然，这样的名称对于单个应用程序来说是唯一的，因此，如果决定将文件移动到远程配置服务器中，则应该注意它们的名称。客户端应用程序名称是从 spring.application.name 注入的，在这种情况下，它是 client-service。因此，总而言之，笔者已经在 src/main/resources/config 目录中创建了名为 client-service-zone[n].yml 的 3 个配置文件，其中，[n]是实例的编号。现在，当开发人员调用 http://localhost:8888/client-service/zone1 端点时，将接收到以下 JSON 格式的响应。

```
{
 "name":"client-service",
 "profiles":["zone1"],
 "label":null,
 "version":null,
 "state":null,
 "propertySources":[{
   "name":"classpath:/config/client-service-zone1.yml",
   "source":{
     "eureka.instance.metadataMap.zone":"zone1",
"eureka.client.serviceUrl.defaultZone":"http://localhost:8761/eureka/",
     "server.port":"${PORT:8081}"
   }
 }]
}
```

还可以为第二个实例调用 http://localhost:8888/client-service-zone2.properties，它将以下响应作为属性列表返回。

```
eureka.client.serviceUrl.defaultZone: http://localhost:8762/eureka/
eureka.instance.metadataMap.zone: zone2
server.port: 8082
```

HTTP API 端点的最后一个可用版本 http://localhost:8889/client-service-zone3.yml 将返回与输入文件相同的数据。以下是第三个实例的结果。

```
eureka:
 client:
   serviceUrl:
     defaultZone: http://localhost:8763/eureka/
 instance:
   metadataMap:
     zone: zone3
server:
 port: 8083
```

## 5.2　构建服务器端应用程序

在第 5.1 节中已经讨论了 HTTP API（一种由 Spring Cloud Config Server 提供的基于资源的 API），以及通过它创建和存储属性的方法。现在我们需要回归到基础方法。与发现服务器相同，Config Server 可以作为 Spring Boot 应用程序运行。要在服务器端启用它，应该在 pom.xml 文件的依赖项中包含 spring-cloud-config-server。

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-config-server</artifactId>
</dependency>
```

除此之外，开发人员还应该在主应用程序类上启用 Config Server。将服务器端口更改为 8888 是一个好主意，因为客户端上 spring.cloud.config.uri 属性的默认值也是 8888 端口。当然，这里说的是它在客户端上自动配置时的情况。要将服务器切换到不同的端口，则应该将 server.port 属性设置在 8888 端口上，或者使用 spring.config.name= configserver 属性来启动它。下面是一个在 spring-cloud-config-server 库中嵌入的 configserver.yml。

```
@SpringBootApplication
@EnableConfigServer
public class ConfigApplication {
```

```
 public static void main(String[] args) {
   new
SpringApplicationBuilder(ConfigApplication.class).web(true).run(args);
 }

}
```

## 5.3　构建客户端应用程序

如果将端口 8888 设置为服务器的默认端口，则客户端的配置就会变得非常简单。开发人员需要做的就是提供包含应用程序名称的 bootstrap.yml 文件，并在 pom.xml 中包含以下依赖项。当然，该规则仅适用于 localhost，因为客户端自动配置的 Config Server 地址就是 http://localhost:8888。

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-config</artifactId>
</dependency>
```

如果为服务器设置了不同于 8888 的端口，或者它在与客户端应用程序不同的计算机上运行，则还应在 bootstrap.yml 中设置其当前地址。以下是 Bootstrap 的上下文设置，它允许开发人员从端口 8889 上可用的服务器获取 client-service 的属性。当使用 --spring.profiles.active=zone1 参数运行该应用程序时，它会自动获取配置服务器中为 zone1 配置文件设置的属性。

```
spring:
 application:
   name: client-service
 cloud:
   config:
     uri: http://localhost:8889
```

开发人员可能已经注意到，客户端属性中存在发现服务网络位置的地址。因此，在启动客户端服务之前，应该运行 Eureka Server。当然，Eureka 也有它自己的配置，并且已经存储在 application.yml 文件中，这在第 4 章的示例中已有说明。与 client-service 类似，该配置已分为 3 个配置文件，其中，每个配置文件在服务器的 HTTP 端口的数量和要与之通信的发现对等体列表等属性上都不同。

现在，可以将这些 property 文件放在配置服务器上。Eureka 将在启动时获取分配给

所选配置文件的所有设置。文件命名与已描述的标准一致，这意味着它应该是 discovery-service-zone[n].yml。在运行 Eureka Server 之前，应该在依赖项中包含 spring-cloud-starter-config 以启用 Spring Cloud Config Client，并将 application.yml 替换为 bootstrap.yml，如下所示。

```
spring:
 application:
   name: discovery-service
 cloud:
   config:
     uri: http://localhost:8889
```

接下来，可以通过在--spring.profiles.active 属性中设置不同的配置文件名称，以对等通信模式运行 Eureka Server 的 3 个实例。启动 3 个客户端服务实例后，我们的架构将如图 5.1 所示。与第 4 章中的示例相比，这里的客户端和发现服务都从 Spring Cloud Config Server 获取配置，而不是将其保存为胖 JAR 中的 YML 文件。

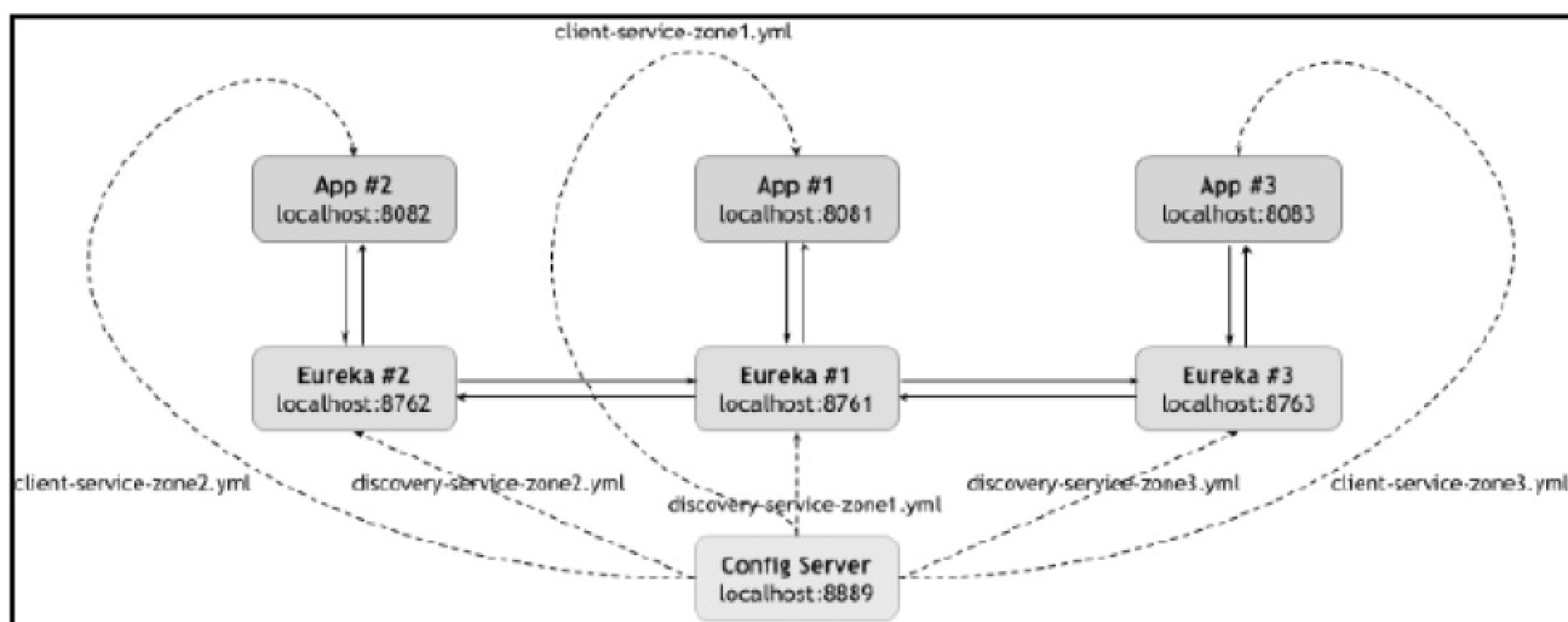



图 5.1　启动 3 个客户端服务实例之后的架构

## 5.4　客户端引导方法

在之前介绍的示例解决方案中，所有应用程序都必须保持配置服务器的网络位置。服务发现的网络位置将作为属性存储在其中。在这一点上，开发人员面临着一个需要讨论的有趣问题。这个问题是：我们的微服务是否应该知道 Config Server 的网络地址？在

之前的讨论中我们同意这个结论：对于所有服务的网络位置来说，应保留的主要位置是服务发现服务器。配置服务器和其他微服务一样，也是 Spring Boot 应用程序，因此从逻辑上讲，它应该向 Eureka 注册它自己，以便为必须从 Spring Cloud Config Server 获取数据的其他服务启用自动发现机制。这反过来要求将服务发现连接设置放在 bootstrap.yml 中，而不是放在 spring.cloud.config.uri 属性中。

在这两种不同方法之间进行选择，是开发人员在设计系统架构时需要做出的决策之一。这里的意思并不是说一个解决方案就一定比另外一个解决方案更好。对于任何使用 spring-cloud-config-client 工件的应用程序来说，其默认行为在 Spring Cloud 术语中被称为配置优先引导（Config First Bootstrap）。当配置客户端启动时，它会绑定到服务器并使用远程属性源初始化上下文。这个解决方案在本章介绍的第一个示例中已经有所体现。在第二个解决方案中，Config Server 将注册服务发现，并且所有应用程序都可以使用 DiscoveryClient 来定位它。这种方法被称为发现优先引导（Discovery First Bootstrap）。接下来我们将通过一个具体示例来说明这个概念。

要访问 GitHub 上的这个示例，开发人员需要切换到 config_with_discovery 分支。以下是其链接地址。

```
https://github.com/piomin/sample-spring-cloud-netflix/tree/config with
discovery.
```

第一个更改与 sample-service-discovery 模块有关。在本示例中，开发人员不需要 spring-cloud-starter-config 依赖，因为其简单配置并不是从远程属性源获取，而是在 bootstrap.yml 中设置的。与前面的示例不同，这里我们启动了一个独立的 Eureka 实例，以简化练习。

```
spring:
 application:
   name: discovery-service

server:
 port: ${PORT:8761}

eureka:
 client:
   registerWithEureka: false
   fetchRegistry: false
```

与前面的示例相比还有一个不同，那就是开发人员应该为 Config Server 包含 spring-cloud-starter-eureka 依赖项。现在，完整的依赖项列表显示在以下代码中。此外，

必须通过在主类上声明@EnableDiscoveryClient 注释来启用发现客户端，并且应通过在 application.yml 文件中将 eureka.client.serviceUrl.defaultZone 属性设置为 http://localhost:8761/eureka/来提供 Eureka 服务器地址。

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-config-server</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-eureka</artifactId>
</dependency>
```

在客户端应用程序中，不再需要保存配置服务器的地址，唯一要做的事情便是设置服务 ID，以防它与 Config Server 不同。根据以上示例中用于服务的命名约定，该 ID 应该是 config-server。它应该被 spring.cloud.config.discovery.serviceId 属性覆盖。为了允许发现机制启用，使得发现机制可以从配置服务器获取远程属性源，开发人员应该设置 spring.cloud.config.discovery.enabled=true。

```
spring:
 application:
   name: client-service
 cloud:
   config:
     discovery:
       enabled: true
       serviceId: config-server
```

图 5.2 是包含 Eureka 仪表板的屏幕截图，其中有一个 Config Server 的实例和 3 个已经注册的 client-service 实例。该客户端的 Spring Boot 应用程序的每一个实例都与上一个示例相同，并且使用了-- spring.profiles.active=zone[n]参数启动，其中，n 是区域的编号。唯一的区别是 Spring Cloud Config Server 提供的所有客户端服务配置文件都具有与 Eureka Server 相同的连接地址。

Instances currently registered with Eureka

| Application | AMIs | Availability Zones | Status |
|---|---|---|---|
| CLIENT-SERVICE | n/a (3) | (3) | UP (3) - minkowp-l.p4.org:client-service:8082 , minkowp-l.p4.org:client-service:8081 , minkowp-l.p4.org:client-service:8083 |
| CONFIG-SERVER | n/a (1) | (1) | UP (1) - minkowp-l.p4.org:config-server:8889 |

图 5.2　Eureka 仪表板屏幕截图

# 5.5　存储库后端类型

在本章中，之前的所有示例都使用了文件系统后端，这意味着配置文件是从本地文件系统或类路径加载的。这种类型的后端非常适合教程或测试。但是，如果要在生产模式中使用 Spring Cloud Config，则需要考虑其他选项。第一个是基于 Git 的存储库后端（Repository Backend），并且在默认情况下也会启用它。当然，它并不是唯一可用作配置源存储库的版本控制系统（Version Control System，VCS）。

另一个选项是 SVN（它是 Subversion 的简称，这也是一个开放源代码的版本控制系统）。或者，开发人员还可以考虑创建一个复合环境，它可能同时包含 Git 和 SVN 存储库。下一个受支持的后端类型是 Vault，它基于由 HashiCorp 提供的工具。在管理密码或证书等安全属性时尤其有用。接下来我们将详细讨论每一个解决方案。

## 5.5.1　文件系统后端

关于文件系统后端的内容，其实在前面的示例中已经讨论过很多了。所有这些示例都展示了如何在类路径中存储属性源，并且还可以从磁盘加载它们。默认情况下，Spring Cloud Config Server 会尝试在应用程序的工作目录或此位置的 config 子目录中查找文件。开发人员可以使用 spring.cloud.config.server.native.searchLocations 属性覆盖默认位置。搜索位置路径可能包含 application、profile 和 label 等占位符。如果在位置路径中没有使用任何占位符，则存储库会自动将 label 参数附加为后缀。

因此，配置文件将从每个搜索位置和与 label 参数同名的子目录加载。这意味着 file:/home/example/config 与 file:/home/example/config、file:/home/example/config/{label}是相同的。通过将 spring.cloud.config.server.native.addLabelLocations 设置为 false 可以禁用此行为。

正如前文所述，文件系统后端不是生产部署的好选择。如果将属性源放在 JAR 文件内的类路径中，则每次更改都需要重新编译应用程序。另外，使用 JAR 之外的文件系统虽然不需要重新编译，但如果在高可用性模式下有多个配置服务实例，则此方法可能会很麻烦。在这种情况下，需要跨所有实例共享文件系统或保存每个运行实例的所有属性源的副本。Git 后端就没有这些缺点，这也是它被推荐用于生产模式的原因。

## 5.5.2　Git 后端

Git 版本控制系统具有的一些功能，使其作为属性源的存储库非常有用。它允许开发人员轻松管理和审核更改。通过使用众所周知的 VCS 机制，如提交（Commit）、还原（Revert）和分支（Branch），开发人员可以比使用文件系统方法更轻松地执行重要操作。这种类型的后端还有另外两个关键优势。它会强制在 Config Server 源代码和 property 文件存储库之间产生分离。如果开发人员再看一遍前面的示例，就会发现这些示例的 property 文件是与应用程序源代码一起存储的。

可能有些人会说，即使开发人员使用的是文件系统后端，也可以将整个配置作为一个单独的项目存储在 Git 上，并根据需要将其上传到远程服务器。这样说固然没错，但是，如果能与 Spring Cloud Config 一起使用 Git 后端，则可以使用现成的有效机制，何乐而不为呢？此外，它还解决了与运行多个服务器实例相关的问题。如果使用的是远程 Git 服务器，则可以在所有正在运行的实例中轻松共享更改。

### 1．不同的协议

要为应用程序设置 Git 存储库的位置，可以使用 application.yml 中的 spring.cloud.config.server.git.uri 属性。如果开发人员熟悉 Git，则应该很清楚地知道，可以使用 file、http/https 和 ssh 协议实现克隆。对本地存储库的访问允许开发人员在没有远程服务器的情况下快速开始。它可以使用 file 前缀进行配置，如 spring.cloud.config.server.git.uri=file:/home/git/config-repo。要在高可用性模式下运行 Config Server（这是更高级的用法），则可以使用 SSH 或 HTTPS 远程协议。在这种情况下，Spring Cloud Config 将克隆远程存储库，并在此基础上使用本地工作副本作为缓存。

### 2．在 URI 中使用占位符

这里还支持所有最近列出的占位符，如 application、profile 和 label。开发人员可以使用占位符为每个应用程序创建一个单独的存储库，如 https://github.com/piomin/{application}，甚至可以为每个配置文件创建单独的存储库，如 https://github.com/piomin/{profile}。这种类型的后端实现可以将 HTTP 资源的 label 参数映射到 Git 标签，该标签可以引用提交 ID、分支或标记名称等。同样，要理解这些有趣功能的最合适的方法就是通过示例，所以接下来我们将创建一个专门用于存储应用程序属性源的 Git 存储库。

### 3．构建服务器应用程序

笔者已经创建了一个示例配置存储库，它可以在以下 GitHub 地址找到：

```
https://github.com/piomin/sample-spring-cloud-config-repo.git.
```

在该存储库中放置了本章第一个示例所使用的所有属性源，其中说明了在不同发现区域中运行的客户端应用程序的原生配置文件支持。现在，该存储库保存了此列表中可见的文件，如图 5.3 所示。

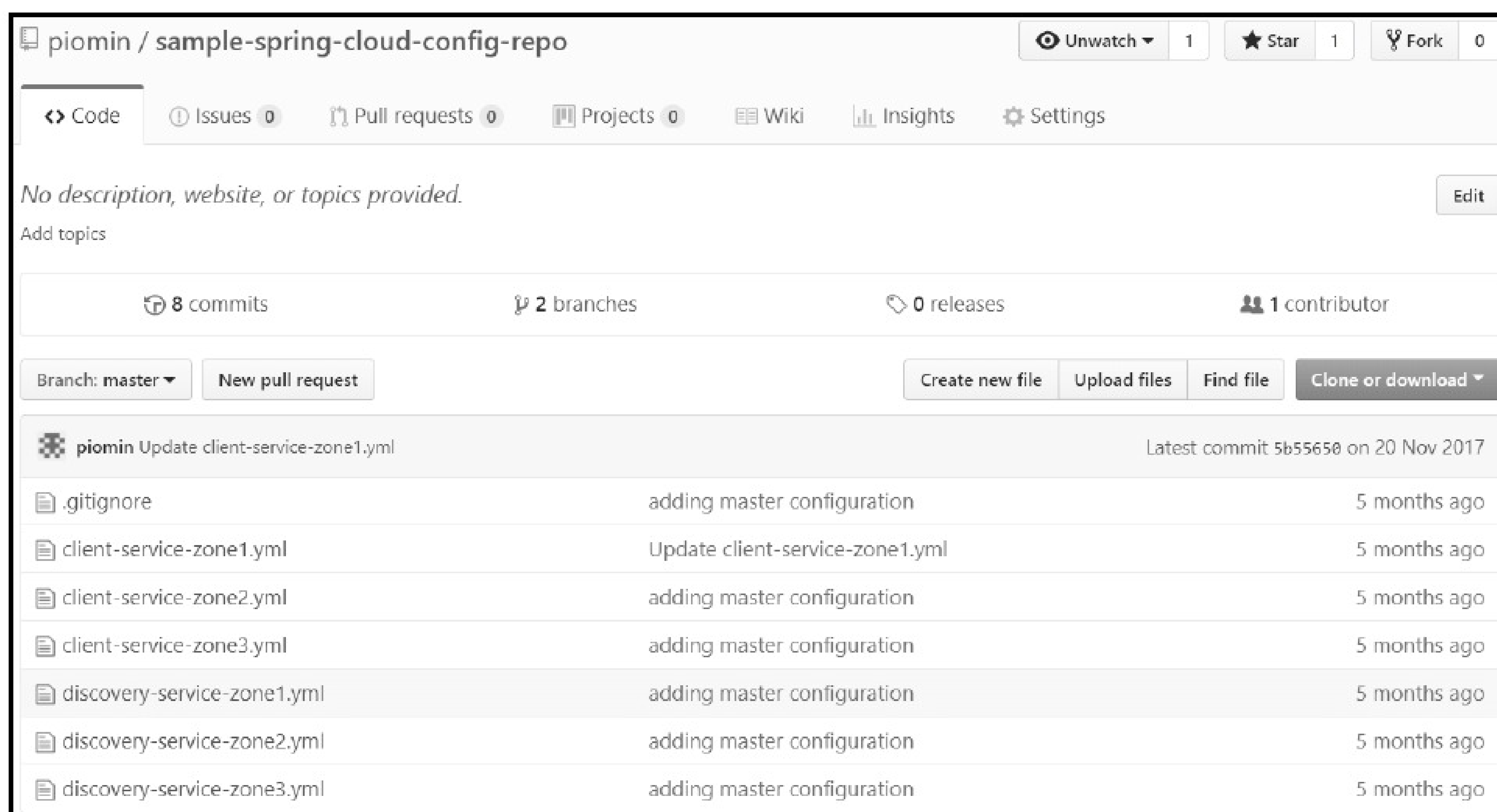



图 5.3　GitHub 上的存储库示例

Spring Cloud Config Server 默认会在第一次 HTTP 资源调用后尝试克隆存储库。如果要在启动后强制克隆它，则应将 cloneOnStart 属性设置为 true。除此之外，还需要设置存储库连接设置和账户身份验证凭据。

```
spring:
 application:
   name: config-server
 cloud:
   config:
     server:
       git:
         uri:https://github.com/piomin/sample-spring-cloud-config-repo.git
         username: ${github.username}
```

```
          password: ${github.password}
          cloneOnStart: true
```

运行该服务器之后，我们可以调用之前练习中已知的端点，如 http://localhost:8889/client-service/zone1 或 http://localhost:8889/client-service-zone2.yml。其结果与前面的测试结果相同，唯一的区别在于数据源。

现在可以来进行另一项练习。在第 5.4 节的示例中，由于采用了发现优先引导方法，并且启用了 native 配置文件，所以该示例需要对客户端的属性略作修改。现在因为使用的是 Git 后端，所以，可以针对这种情况开发更智能的解决方案。在当前的方法中，开发人员应该在 GitHub 上的配置库中创建 discovery 分支（https://github.com/piomin/sample-spring-cloud-config-repo/tree/discovery），然后再把专用于该应用程序的文件放上去（该应用程序将演示发现优先引导机制）。如果将 label 参数设置为 discovery 然后再调用 Config Server 端点，则它将从新分支中获取数据。开发人员可以尝试分别调用 http://localhost:8889/client-service/zone1/discovery 和 http://localhost:8889/discovery/client-service-zone2.yml，然后检查和对比其结果。

现在来考虑另一种情况。假设已经更改了 client-service 第三个实例的服务器端口，但由于某种原因，想要回到之前的值。那么，是否必须更改并提交 client-service-zone3.yml 才能使用先前的端口值呢？答案是不必，现在开发人员所要做的就是在调用 HTTP API 资源时将提交 ID 作为 label 参数传递。已经执行的更改如图 5.4 所示。




图 5.4　通过传递 label 参数方式修改的端口

如果使用父提交 ID 而不是分支名称调用 API 端点，则将返回旧端口号作为响应。图 5.4 是调用 http://localhost:8889/e546dd6/client-service-zone3.yml 的结果，其中，e546dd6 就是以前提交的 ID。

```
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
  instance:
```

```
     metadataMap:
       zone: zone3
server:
  port: 8083
```

### 4. 客户端配置

在使用 Git 后端构建服务器端时，仅演示了 HTTP 资源调用的示例。以下是客户端应用程序的示例配置。开发人员也可以在 spring.profiles.active 运行参数中传递它，而不是在 bootstrap.yml 中设置 profile 属性。此配置将使客户端可以从 discovery 分支获取属性。开发人员也可以决定通过在 label 属性中设置它来切换到某个提交 ID，这在前面已经介绍过。

```
spring:
 application:
   name: client-service
 cloud:
   config:
     uri: http://localhost:8889
     profile: zone1
     label: discovery
#    label: e546dd6 //取消代码注释即可回滚
```

### 5. 多个存储库

开发人员有时可能需要为单台配置服务器配置多个存储库。例如，可以想到的一种情况是：必须将业务配置与典型技术配置分开，这绝对是有可能的。

```
spring:
 cloud:
   config:
     server:
       git:
         uri:
https://github.com/piomin/spring-cloud-config-repo/config-repo
         repos:
           simple: https://github.com/simple/config-repo
           special:
             pattern: special*/dev*,*special*/dev*
             uri: https://github.com/special/config-repo
           local:
             pattern: local*
             uri: file:/home/config/config-repo
```

### 5.5.3　Vault 后端

前文已经介绍过，Vault 是一种工具，它可以通过统一的接口对机密信息进行安全访问。为了使 Config Server 能够使用该类型的后端，开发人员必须使用 Vault 配置文件--spring.profiles.active=vault 运行它。当然，在运行 Config Server 之前，还需要安装并启动 Vault 实例。建议开发人员使用 Docker 来完成该任务。由于这是本书第一次介绍与 Docker 相关的内容，并不是每个人都知道该工具，在本书第 14 章“Docker 支持”中简要介绍了 Docker，并提供了它的基本命令和用例，因此，如果这是你第一次接触该技术，则不妨先跳到第 14 章翻看一下其内容。对于那些熟悉 Docker 的人来说，则应该很容易理解以下命令示例，它将在开发模式下运行 Vault 容器。开发人员可以使用 VAULT_DEV_LISTEN_ADDRESS 参数覆盖默认的侦听地址，或者使用 VAULT_DEV_ROOT_TOKEN_ID 参数覆盖初始生成的根令牌的 ID。

```
docker run --cap-add=IPC_LOCK -d --name=vault -e
'VAULT_DEV_ROOT_TOKEN_ID=client' -p 8200:8200 vault
```

#### 1. Vault 入门

Vault 提供了一个命令行接口，可用于向服务器添加新值并从服务器读取它们。以下显示了调用这些命令的示例。但是，由于我们将 Vault 作为 Docker 容器运行，因此，管理机密最方便的方法是通过 HTTP API。

```
$ vault write secret/hello value=world
$ vault read secret/hello
```

HTTP API 可用于我们的 Vault 实例（位于 http://192.168.99.100:8200/v1/secret 地址下）。调用该 API 的每个方法时，需要将令牌 X-Vault-Token 作为请求头传递。因为我们在启动 Docker 容器时已经在 VAULT_DEV_ROOT_TOKEN_ID 环境参数中设置了该值，所以它等于 client。否则，它将在启动期间自动生成，并且可以通过调用命令 docker logs vault 从日志中读取。要开始使用 Vault，开发人员实际上需要了解两种 HTTP 方法——POST 和 GET。在调用 POST 方法时，可以定义应该添加到服务器的机密列表。在以下 curl 命令示例中，它所传递的参数就是使用 kv（key/value）后端创建的，该后端的作用类似于键/值（Key/Value）存储。

```
$ curl -H "X-Vault-Token: client" -H "Content-Type: application/json" -X
POST -d '{"server.port":8081,"sample.string.property": "Client
```

```
App","sample.int.property": 1}'
http://192.168.99.100:8200/v1/secret/client-service
```

可以使用 GET 方法从服务器读取新添加的值。

```
$ curl -H "X-Vault-Token: client" -X GET
http://192.168.99.100:8200/v1/secret/client-service
```

### 2. 与 Spring Cloud Config 集成

正如前文所述，开发人员必须使用--spring.profiles.active=vault 参数运行 Spring Cloud Config Server，这样才能启用 Vault 作为后端存储。要覆盖默认的自动配置设置，应该在 spring.cloud.config.server.vault.* 键下面定义属性。以下示例显示了我们的示例应用程序的当前配置。在 GitHub 上也提供了这样一个示例应用程序，开发人员可以切换到 config_vault 分支（https://github.com/piomin/sample-spring-cloud-netflix/tree/config_vault）来访问它。

```
spring:
 application:
   name: config-server
 cloud:
   config:
     server:
       vault:
         host: 192.168.99.100
         port: 8200
```

现在，开发人员可以调用由 Config Server 公开的端点。虽然仍必须在请求头中传递令牌，但这一次它的名称是 X-Config-Token。

```
$ curl -X "GET" "http://localhost:8889/client-service/default" -H "X-
Config-Token: client"
```

其响应结果应该如下所示。这些属性是客户端应用程序的配置文件的默认属性。开发人员还可以添加所选配置文件的特定设置，方法是在逗号字符后面使用配置文件名称，以调用 Vault HTTP API 方法，如 http://192.168.99.100:8200/v1/secret/client-service, zone1。如果调用路径中包含此类配置文件名称，则响应中将返回 default 和 zone1 配置文件的属性。

```
{
    "name":"client-service",
    "profiles":["default"],
```

```
    "label":null,
    "version":null,
    "state":null,
    "propertySources":[{
        "name":"vault:client-service",
        "source":{
            "sample.int.property":1,
            "sample.string.property":"Client App",
            "server.port":8081
        }
    }]
}
```

### 3. 客户端配置

使用 Vault 作为 Config Server 的后端时，客户端需要为服务器传递令牌，以便能够从 Vault 检索值。应使用 bootstrap.yml 文件中的 spring.cloud.config.token 属性在客户端配置设置中提供此令牌。

```
spring:
 application:
   name: client-service
 cloud:
   config:
     uri: http://localhost:8889
     token: client
```

# 5.6 其 他 功 能

现在我们来看一看 Spring Cloud Config 其他一些有用的功能。

## 5.6.1 启动失败和重试

有时，如果 Config Server 不可用，则启动该应用程序没有任何意义。在这种情况下，开发人员可能会想要暂停一个有异常的客户端。为此，必须将引导配置属性 spring.cloud.config.failFast 设置为 true。但是，这种比较激进的解决方案有时候并不是开发人员想要的。如果仅仅是偶尔无法访问配置服务器，则更好的方法是继续尝试重新连接，直到成功为止。spring.cloud.config.failFast 属性仍然必须等于 true，但开发人员还需要将 spring-

retry 库和 spring-boot-starter-aop 添加到应用程序类路径中。默认行为将假定重试 6 次，初始退避（Backoff）重试间隔为 1000 毫秒。开发人员可以使用 spring.cloud.config.retry.* 配置属性覆盖这些设置。

### 5.6.2　保护客户端的安全

与服务发现相同，开发人员可以使用基本身份验证来保护配置服务器。使用 Spring Security 可以在服务器端轻松启用它。在这种情况下，客户端仅需设置 bootstrap.yml 文件中的用户名和密码。

```
spring:
 cloud:
   config:
     uri: https://localhost:8889
     username: user
     password: secret
```

## 5.7　自动重新加载配置

前文已经讨论过 Spring Cloud Config 最重要的功能。现在我们可以实现一个示例，并通过它来演示如何将不同的后端存储用作存储库。但是，无论开发人员决定选择的是文件系统、Git 还是 Vault，客户端应用程序都需要重新启动才能从服务器端获取最新配置。当然，本示例有时候并不是最佳解决方案，特别是如果有许多微服务运行，并且其中一些使用相同的通用配置的话，则更是如此。

### 5.7.1　解决方案架构

即使开发人员已经为每一个应用程序创建了一个专用的 property 文件，如果能有机会动态重新加载它而不必重启，那么这也是很有帮助的。很多人可能已经想到了，既然 Spring Boot 可以有这样的解决方案，那么对于 Spring Cloud 来说自然也是适用的。本书第 4 章“服务发现”在解释从服务发现服务器注销时，曾经引入了一个端点/shutdown，它可以用于从容地关闭。

还有一个可用于 Spring 环境重启的端点，其工作方式与/shutdown 类似。

客户端上的端点只是一个大得多的系统的组件，而该系统需要被包含才能启用 Spring Cloud Config 的推送通知。最流行的源代码存储库提供商（如 GitHub、GitLab 和 Bitbucket）

能够通过提供 WebHook 机制发送有关存储库中更改的通知。开发人员可以使用提供商的 Web 仪表板将 WebHook 配置为 URL 和所选事件类型的列表。这样，提供商就可以使用包含提交列表的正文调用 WebHook 中定义的 POST 方法。在项目中需要包含 Spring Cloud Bus 依赖项，以便在 Config Server 端启用监控端点。当由于 WebHook 激活而调用此端点时，Config Server 会准备并发送一个事件，其中包含已通过最近一次提交而修改的属性源列表。该事件将被发送给消息代理。Spring Cloud Bus 为 RabbitMQ 和 Apache Kafka 提供了实现。对于 RabbitMQ 来说，可以通过包括 spring-cloud-starter-bus-amqp 依赖项来启用该项目；而对于 Apache Kafka 来说，可以通过包括 spring-cloud-starter-bus-kafka 依赖项来启用。此外，还应为客户端应用程序声明这些依赖项，才能从消息代理那里接收到消息。开发人员还应该通过使用@RefreshScope 注解所选的配置类来启用客户端上的动态刷新机制。此解决方案的架构如图 5.5 所示。

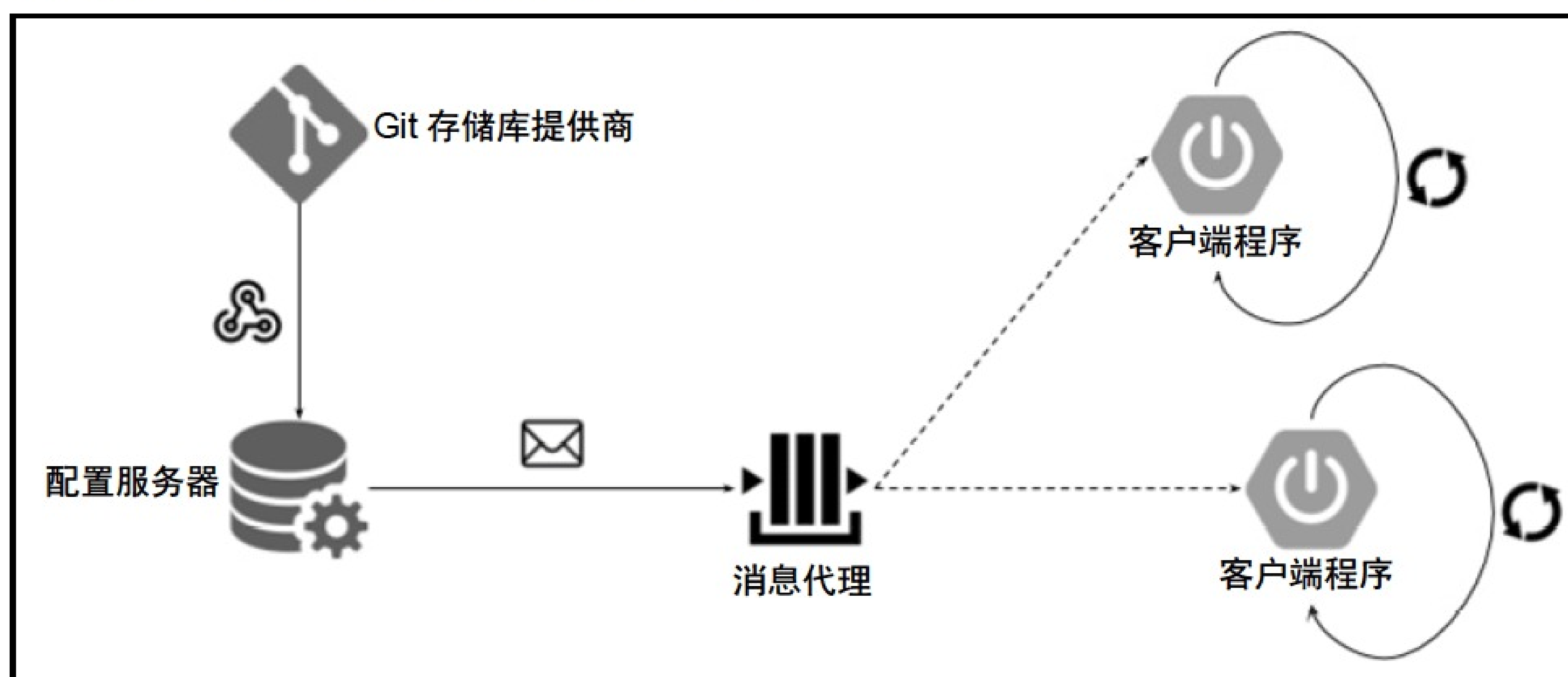



图 5.5　解决方案架构

## 5.7.2　使用@RefreshScope 重新加载配置

这次异常将从客户端开始。示例应用程序可在 GitHub（https://github.com/piomin/sample-spring-cloud-config-bus.git）上获得。与前面的示例相同，它使用了 Git 存储库作为后端存储，这也可以在 GitHub（https://github.com/piomin/sample-spring-cloud-config-repo）上创建。此外，我们还向客户端的配置文件添加了一些新属性，并将更改提交到存储库。以下是客户端配置的当前版本。

```
eureka:
 instance:
```

```
    metadataMap:
      zone: zone1
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
server:
  port: ${PORT:8081}
management:
  security:
    enabled: false
sample:
  string:
    property: Client App
  int:
    property: 1
```

开发人员可以通过将 management.security.enabled 设置为 false 来禁用 Spring Boot Actuator 端点的安全性，这是在不传递安全凭证的情况下调用这些端点所必需的。此外还需要添加 sample.string.property 和 sample.int.property 两个测试参数，以根据示例中的值来演示 bean 刷新机制。Spring Cloud 为 Spring Boot Actuator 提供了一些额外的 HTTP 管理端点。其中一个是/refresh，它负责重新加载 Bootstrap 上下文和使用@RefreshScope 注解的刷新 bean。这是一个 HTTP POST 方法，可以在客户端实例（http://localhost:8081/refresh）上调用。在测试该功能之前，开发人员需要运行发现和配置服务器。应使用--spring.profiles.active=zone1 参数启动客户端应用程序。在以下类中，可以将测试属性 sample.string.property 和 sample.int.property 注入字段中。

```
@Component
@RefreshScope
public class ClientConfiguration {
  @Value("${sample.string.property}")
  private String sampleStringProperty;
  @Value("${sample.int.property}")
  private int sampleIntProperty;

  public String showProperties() {
    return String.format("Hello from %s %d", sampleStringProperty,
sampleIntProperty);
  }

}
```

该 bean 被注入 ClientController 类并在 ping 方法中调用，该方法在 http://localhost:8081/ping 位置处公开。

```
@RestController
public class ClientController {

 @Autowired
 private ClientConfiguration conf;

 @GetMapping("/ping")
 public String ping() {
   return conf.showProperties();
 }

}
```

现在，开发人员可以改变 client-service-zone1.yml 中测试属性的值并提交它们。如果调用 Config Server HTTP 端点/client-service/zone1，则将看到作为响应返回的最新值。但是，当调用客户端应用程序上公开的/ping 方法时，开发人员仍会在如图 5.6 所示的屏幕截图的左侧看到较旧的值。为什么？其实这也很好理解，虽然 Config Server 会自动检测存储库更改，但是客户端应用程序无法在没有任何触发器的情况下自动刷新。它需要重新启动才能读取最新设置，或者开发人员也可以通过调用之前所描述的/refresh 方法来强制重新加载配置。




图 5.6　在没有任何触发器的情况下客户端应用程序不会自动刷新

在客户端应用程序上调用/refresh 端点后，开发人员将在日志文件中看到已经重新加载配置。现在，如果再次调用/ping，则会在响应中返回最新的属性值，如图 5.7 所示。该示例说明了热重载如何适用于 Spring Cloud 应用程序，但它显然不是我们的目标解决方案。我们的下一步是启用与消息代理的通信。

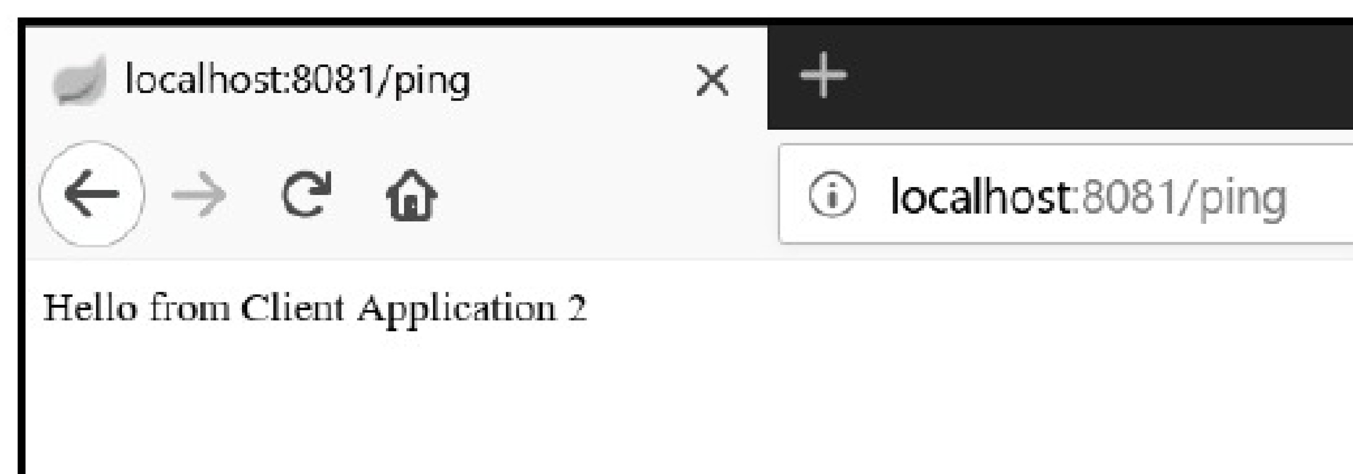



图 5.7　调用/ping 之后的响应结果

### 5.7.3　使用来自消息代理的事件

前文已经提到过，开发人员可以选择两个与 Spring Cloud Bus 集成的消息代理（Message Broker）。在本示例中，将演示如何运行和使用 RabbitMQ。关于这个解决方案有必要多做一些介绍，因为这是在本书中第一次处理它。RabbitMQ 已经发展成为最受欢迎的消息代理软件。它是用 Erlang 编写的，并且实现了高级消息队列协议（Advanced Message Queueing Protocol，AMQP）。它易于使用和配置，即使对于我们正在讨论的集群或高可用性等机制来说也不例外。

要在开发人员的机器上运行 RabbitMQ，最简便的方法是通过 Docker 容器。该容器已经公开了两个端口。第一个端口用于客户端连接（5672），第二个端口用于管理仪表板（15672）。此外，我们还使用了 management 标记来运行镜像（Image），以启用用户界面仪表板，这在默认版本中是不可用的。

```
docker run -d --name rabbit -p 5672:5672 -p 15672:15672 rabbitmq:management
```

要为我们的示例客户端应用程序启用对于 RabbitMQ 代理的支持，应该在 pom.xml 中包含以下依赖项。

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-bus-amqp</artifactId>
</dependency>
```

该库包含自动配置设置。因为我们是在 Windows 系统上运行 Docker，所以需要覆盖一些默认属性。完整的服务配置存储在 Git 存储库中，因而更改仅影响远程文件。开发人员应该将以下参数添加到以前使用的客户端属性源版本中。

```
spring:
 rabbitmq:
   host: 192.168.99.100
   port: 5672
```

```
  username: guest
  password: guest
```

此时如果运行该客户端应用程序，则将在 RabbitMQ 中自动创建交换消息和队列。开发人员可以登录 http://192.168.99.100:15672 上的管理仪表板轻松查看。默认用户名和密码是 guest/guest。如图 5.8 所示是笔者的 RabbitMQ 实例的屏幕截图，可以看到已经创建了一个名为 springCloudBus 的交换消息（Exchange），有两个绑定到客户端队列和 Config Server 队列（笔者已经使用更改运行它，详见第 5.7.4 节中的说明）。在目前这个阶段，开发人员还不必着急深入了解 RabbitMQ 及其架构，在本书第 11 章“消息驱动的微服务”有关 Spring Cloud Stream 项目的讨论中将会对此做更详细的说明。

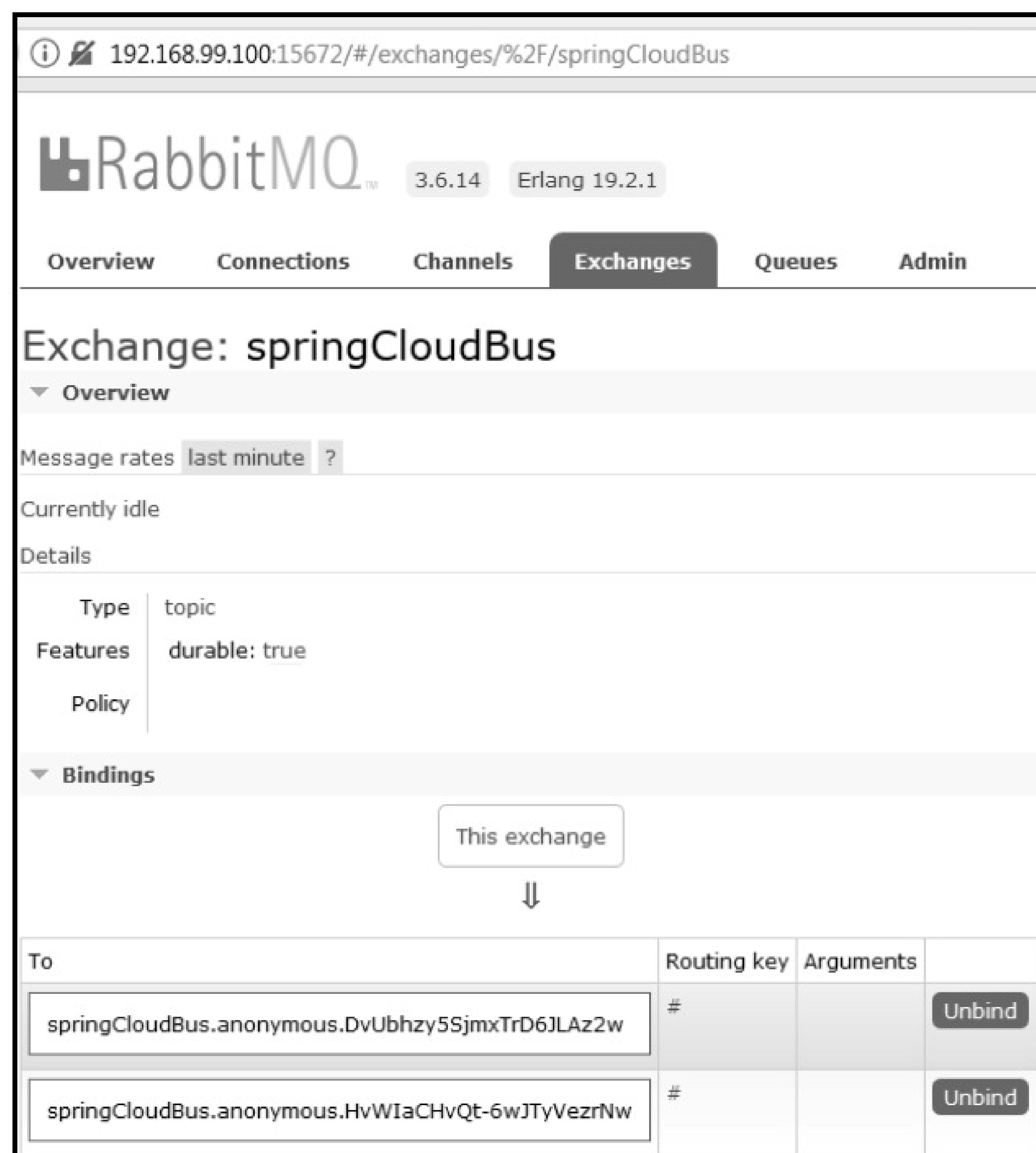



图 5.8　RabbitMQ 实例的屏幕截图

## 5.7.4　监视 Config Server 上的存储库更改

Spring Cloud Config Server 必须在前面描述的过程中执行两项任务。首先，它必须检

测存储在 Git 存储库中的 property 文件中的更改。这可以通过公开特殊端点来实现，该端点将由存储库提供商通过 WebHook 调用。第二个步骤是准备并发送一个针对可能已更改的应用程序的 RefreshRemoteApplicationEvent 事件，这反过来又要求开发人员与消息代理建立连接。spring-cloud-config-monitor 库负责启用/monitor 端点。要启用对 RabbitMQ 代理的支持，应该包含与客户端应用程序相同的启动工件。

```
<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-config-monitor</artifactId>
</dependency>
<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-starter-bus-amqp</artifactId>
</dependency>
```

这还不是全部。开发人员还应在 application.yml 中激活配置监控程序（Monitor）。由于每个存储库提供商都在 Spring Cloud 中具有专用实现，因而有必要选择应该启用哪一个存储库提供商。

```
spring:
 application:
   name: config-server
 cloud:
   config:
     server:
       monitor:
         github:
           enabled: true
```

开发人员可以自定义更改检测机制。默认情况下，它会检测与应用程序名称匹配的文件中的更改。要覆盖该行为，需要提供 PropertyPathNotificationExtractor 的自定义实现。它接受请求头和正文参数，并返回已更改的文件路径列表。为了支持来自 GitHub 的通知，可以使用 spring-cloud-config-monitor 提供的 GithubPropertyPathNotificationExtractor。

```
@Bean
public GithubPropertyPathNotificationExtractor
githubPropertyPathNotificationExtractor() {
    return new GithubPropertyPathNotificationExtractor();
}
```

### 1. 手动模拟更改事件

monitor 端点可以由 Git 存储库提供商（如 GitHub、Bitbucket 或 GitLab）上配置的

WebHook 调用。使用在 localhost 上运行的应用程序测试此类功能很麻烦。事实证明，开发人员可以通过手动调用 POST /monitor 来轻松模拟这样的 WebHook 激活。例如，Github 命令应该在请求中包含标头 X-Github-Event。具有 property 文件更改的 JSON 正文应该如以下 cURL 请求中所示。

```
$ curl -H "X-Github-Event: push" -H "Content-Type: application/json" -X
POST -d '{"commits": [{"modified": ["client-service-zone1.yml"]}]}'
http://localhost:8889/monitor
```

现在，可以在 client-service-zone1.yml 文件中更改并提交一个属性的值，例如，修改属性 sample.int.property。然后，可以使用前面示例命令中显示的参数调用 POST /monitor 方法。如果根据上面的说明配置了所有内容，则应在客户端应用程序端看到以下日志行：Received remote refresh request. Keys refreshed [sample.int.property]（已接收到远程刷新请求。键值已刷新[sample.int.property]）。如果调用客户端微服务公开的/ping 端点，那么它应该返回已更改属性的最新值。

### 2．使用 GitLab 实例在本地进行测试

对于那些不喜欢模拟事件的开发人员，我们提出了一个更实际的练习。但是，需要指出的是，它不仅需要开发技能，还需要掌握持续集成（Continuous Integration）工具的基本知识。我们将首先使用 GitLab 的 Docker 镜像在本地运行 GitLab 实例。GitLab 是一个开源的基于 Web 的 Git 存储库管理器，具有维基百科和问题跟踪的特点。它与 GitHub 或 Bitbucket 等工具非常相似，但可以轻松部署在本地计算机上。

```
docker run -d --name gitlab -p 10443:443 -p 10080:80 -p 10022:22
gitlab/gitlab-ce:latest
```

Web 仪表板可在 http://192.168.99.100:10080 获得。第一步是创建管理员用户，然后使用提供的凭据登录。对于 GitLab 其实不必做过多介绍，它具有用户友好且直观的 GUI 界面，因而开发人员肯定能够轻松使用它。现在继续进行下一步操作，我们已经在 GitLab 中创建了一个名为 sample-spring-cloud-config-repo 的项目，开发人员可以从 http://192.168.99.100:10080/root/sample-spring-cloud-config-repo.git 克隆它。我们在该位置提交了相同的配置文件集，可以在 GitHub 上的示例存储库中找到。下一步是定义一个 WebHook，它将通过推送通知调用 Config Server 的/monitor 端点。要为项目添加新的 WebHook，需要转到 Settings | Integration（设置 | 集成）部分，然后用服务器地址填写 URL 字段（注意，是使用主机名而不是 localhost），然后保持选中 Push events（推送事件）复选框，如图 5.9 所示。

Administrator › sample-spring-cloud-... › Integrations Settings

Integrations

Webhooks can be used for binding events when something is happening within the project.

URL

http://piomin:8889/monitor

Secret Token

Use this token to validate received payloads. It will be sent with the request in the X-Gitlab-Token HTTP header.

Trigger

Push events

This URL will be triggered by a push to the repository

图 5.9　添加新的 WebHook

与使用 GitHub 作为后端存储库提供商的 Config Server 实现相比，我们需要在 application.yml 中更改已启用的 monitor 类型。当然，还需要提供不同的地址。

```
spring:
 application:
   name: config-server
 cloud:
   config:
     server:
       monitor:
         gitlab:
           enabled: true
       git:
         uri:
http://192.168.99.100:10080/root/sample-spring-cloud-config-repo.git
         username: root
         password: root123
         cloneOnStart: true
```

还应该注册另一个 bean 以实现 PropertyPathNotificationExtractor。

```
@Bean
public GitlabPropertyPathNotificationExtractor
gitlabPropertyPathNotificationExtractor() {
    return new GitlabPropertyPathNotificationExtractor();
}
```

最后，开发人员可以在配置文件中进行一些更改，此时 WebHook 应该被激活，并刷新客户端应用程序的配置。

## 5.8　小　　结

本章详细介绍了 Spring Cloud Config 项目的最重要功能。与“服务发现”的介绍方式相同，本章从最基础的并且很简单的客户端和服务器端用例开始，详细讨论了 Config Server 的不同后端存储库类型，并且通过实现示例的方式说明了如何使用文件系统、Git 甚至第三方工具（如 Vault）作为 property 文件的存储库。本章还特别关注与其他组件（如服务发现或大型系统中的多个微服务实例）的互操作性。最后，本章演示了如何在不重启的情况下重新加载应用程序的配置，这是基于 WebHooks 和消息代理实现的。总之，在阅读本章之后，开发人员应该能够将 Spring Cloud Config 用作基于微服务架构的一个元素，并充分利用其主要功能。

在掌握了使用 Spring Cloud 实现“服务发现”和“配置服务器”功能之后，即可进行对服务间通信机制的讨论。在接下来的两章中，我们将分析与其相关的基本示例和一些更高级的应用，这些示例将演示若干个微服务之间的同步通信。

# 第 6 章　微服务之间的通信

在前两章中，我们讨论了与微服务架构——服务发现和配置服务器中非常重要的元素相关的细节。但是，值得一提的是，它们在系统中存在主要是为了帮助管理独立应用程序的整体设置。该管理的一个方面就是微服务之间的通信。在这里，服务发现扮演了一个特别重要的角色，它负责存储所有可用应用程序的网络位置并提供服务。当然，我们也可以设想一个没有服务发现服务器的系统架构，本章就将介绍这样的示例。

当然，参与服务间通信的最重要组件是 HTTP 客户端和客户端负载均衡器。本章将重点关注它们。

本章将要讨论的主题包括：

- 使用 Spring RestTemplate 进行有和没有服务发现的服务间通信。
- 自定义 Ribbon 客户端。
- Feign 客户端提供的主要功能的描述，如与 Ribbon 客户端的集成、服务发现、继承和分区支持。

## 6.1　不同类型的通信

微服务之间的通信具有不同的类型。可以从两个维度来划分它们。第一个维度是将其划分为同步（Synchronous）和异步（Asynchronous）通信协议。异步通信的关键要点是客户端在等待响应时不应该阻塞线程。这种通信最流行的协议是 AMQP，在第 5 章的末尾我们已经介绍了使用该协议的示例。

当然，服务之间的主要通信方式仍然是同步 HTTP 协议，本章将对其进行详细讨论。

第二个维度是按照存在单个消息接收器还是多个消息接收器来划分的，可以划分为“一对一”或“一对多”通信类型。在“一对一”通信中，每个请求仅由一个服务实例处理；在“一对多”通信中，每个请求可以由许多不同的服务处理。本书将在第 11 章“消息驱动的微服务”中对此展开详细的讨论。

## 6.2　使用 Spring Cloud 进行同步通信

Spring Cloud 提供了一组组件来帮助开发人员实现微服务之间的通信。第一个组件是

RestTemplate，它总是用于在客户端使用 RESTful Web 服务，包含在 Spring Web 项目中。要在微服务环境中有效地使用它，应该使用@LoadBalanced 限定符（Qualifier）进行注解。由此它将自动配置为使用 Netflix Ribbon，它将能够通过使用服务名称而不是 IP 地址来利用服务发现。Ribbon 是一个客户端负载均衡器，它提供了一个简单的接口，允许控制 HTTP 和 TCP 客户端的行为。它可以轻松地与其他 Spring Cloud 组件集成，如服务发现或断路器。此外，它对开发人员完全透明。第二个可用组件是 Feign，这是一个同样来自 Netflix OSS 的声明性 REST 客户端。Feign 已经使用 Ribbon 进行负载均衡，并且可以从服务发现中获取数据。可以通过使用@FeignClient 注解方法在界面上轻松声明它。本章将详细介绍此处列出的所有组件。

# 6.3　使用 Ribbon 执行负载均衡

与 Ribbon 有关的主要概念是命名客户端（Client）。这就是为什么开发人员可以使用客户端的名称而不是带有主机名和端口的完整地址来调用其他服务，并且无须连接到服务发现。在这种情况下，应该在 application.yml 文件内的 Ribbon 配置设置中提供地址的列表。

## 6.3.1　使用 Ribbon 客户端启用微服务之间的通信

现在来继续这个示例。它由 4 个独立的微服务组成，其中一些可能会调用其他微服务公开的端点。应用程序源代码可在以下地址处获得。

```
https://github.com/piomin/sample-spring-cloud-comm.git
```

在这个示例中，我们将尝试开发一个简单的订单系统，客户可以在该系统中购买产品。如果客户决定确认要购买的所选产品列表，则 POST 请求将发送到 order-service（订单服务）。它由 REST 控制器内的 Order prepare(@RequestBody Order order){...}方法处理。该方法负责订单的准备。首先，它将从 order-service 中调用适当的 API 方法，考虑产品列表中每个产品的价格、客户订购的历史记录及客户在系统中的类别等因素，计算出最终价格；然后，它通过调用 account-service（账户服务），验证客户的账户余额是否足够执行订单；最后，它返回计算的价格。如果客户确认操作，则调用 PUT /{id}方法。该请求将由 REST 控制器内的 Order accept(@PathVariable Long id){...}方法处理。它会更改订单的状态并从客户的账户中提取资金。该系统架构可以分解为各个微服务，如图 6.1 所示。

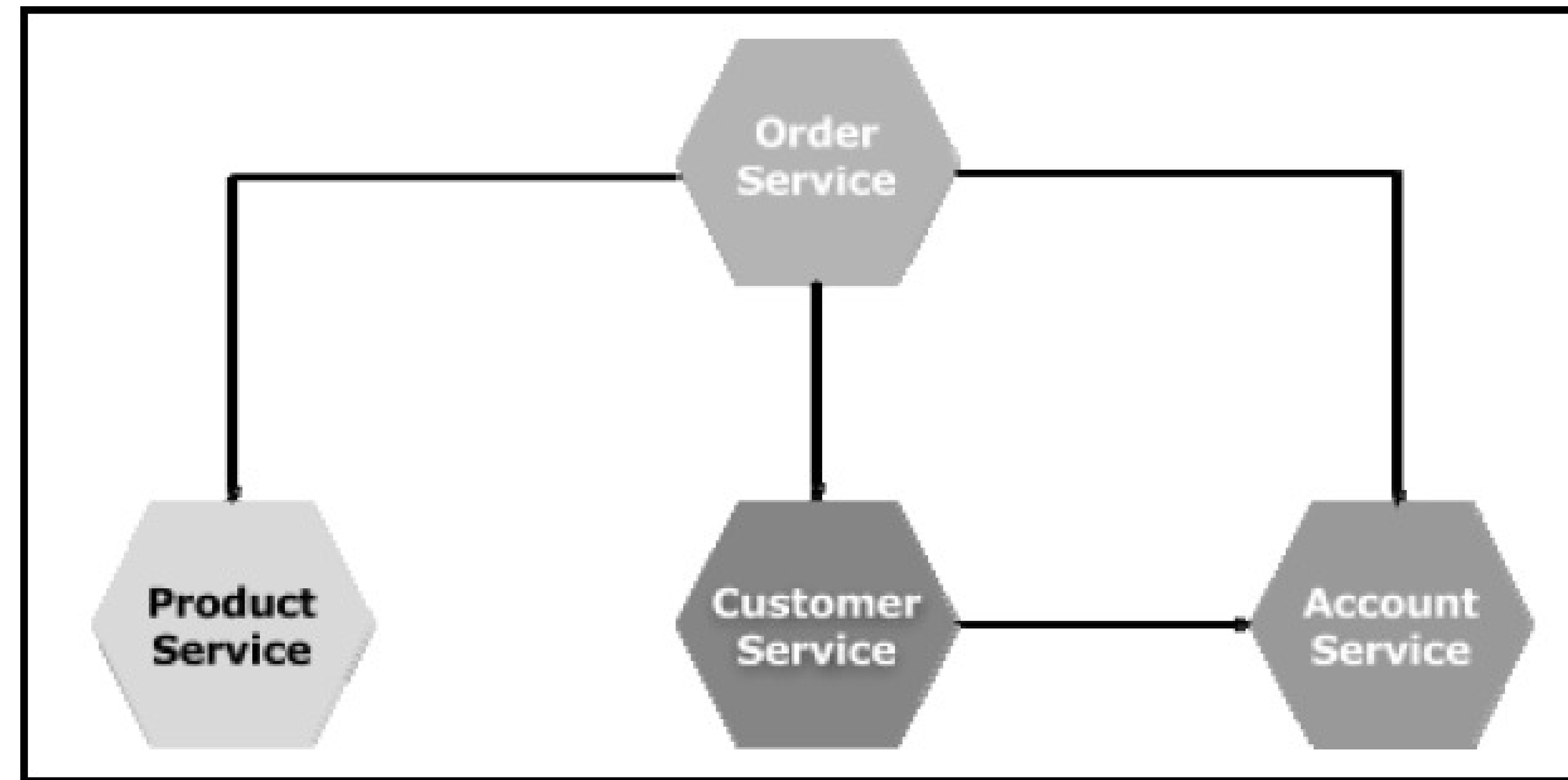



图 6.1　微服务之间通信示例的系统架构

## 6.3.2　静态负载均衡配置

本示例中的 order-service（订单服务）必须与该示例中的所有其他微服务进行通信，以执行所需的操作，因此，开发人员需要使用 ribbon.listOfServers 属性定义 3 个不同的 Ribbon 客户端，并且这些客户端需要包含网络地址设置。该示例中的第二个重要事项是在 Eureka 中禁用默认启用的发现服务。以下是 application.yml 文件中 order-service 的所有已定义属性。

```
server:
 port: 8090

account-service:
 ribbon:
   eureka:
     enabled: false
   listOfServers: localhost:8091
customer-service:
 ribbon:
   eureka:
     enabled: false
   listOfServers: localhost:8092
product-service:
 ribbon:
   eureka:
     enabled: false
   listOfServers: localhost:8093
```

开发人员还应该在项目中包含以下依赖项，以便将 RestTemplate 与 Ribbon 客户端结

合在一起使用。

```
<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-starter-ribbon</artifactId>
</dependency>
<dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

然后，开发人员可以通过声明 application.yml 中配置的名称列表来启用 Ribbon 客户端。为完成此操作，可以使用@RibbonClients 注解 main 类或任何其他 Spring 配置类。开发人员还应该注册 RestTemplate bean 并使用@LoadBalanced 注解它，以启用与 Spring Cloud 组件的交互。

```
@SpringBootApplication
@RibbonClients({
 @RibbonClient(name = "account-service"),
 @RibbonClient(name = "customer-service"),
 @RibbonClient(name = "product-service")
})
public class OrderApplication {

 @LoadBalanced
 @Bean
 RestTemplate restTemplate() {
     return new RestTemplate();
 }

 public static void main(String[] args) {
     new
SpringApplicationBuilder(OrderApplication.class).web(true).run(args);
 }
 // ...
}
```

## 6.3.3　调用其他服务

最后，我们可以开始实现 OrderController（订单控制程序），它负责提供公开到微服

务之外的 HTTP 方法。它注入了 RestTemplate bean 以便能够调用其他 HTTP 端点。开发人员可能会在以下源代码片段中看到使用 application.yml 中配置的 Ribbon 客户端名称而不是 IP 地址或主机名。使用相同的 RestTemplate bean，即可与 3 个不同的微服务进行通信。在这里需要花一点时间来讨论控制程序内部可用的方法。在第一个实现的方法中，可以调用 product-service（产品服务）的 GET 端点，该端点将返回包含所选产品详细信息的列表。然后，可以调用 customer-service（客户服务）公开的 GET /withAccounts/ {id} 方法，它将返回客户详细信息及其账户列表。

现在，开发人员拥有了计算最终订单价格所需的所有信息，并将验证客户在其主账户中是否有足够的资金。PUT 方法将调用 account-service（账户服务）的端点以从客户账户中提取资金。虽然我们花了很多时间来讨论 OrderController 中可用的方法，但我们认为这是必要的，因为相同的示例将用于显示 Spring Cloud 组件的主要功能，这些组件提供了微服务之间同步通信的机制。

```
@RestController
public class OrderController {

 @Autowired
 OrderRepository repository;
 @Autowired
 RestTemplate template;

 @PostMapping
 public Order prepare(@RequestBody Order order) {
      int price = 0;
      Product[] products =
template.postForObject("http://product-service/ids",
order.getProductIds(), Product[].class);
      Customer customer =
template.getForObject("http://customer-service/withAccounts/{id}",
Customer.class, order.getCustomerId());
      for (Product product : products)
          price += product.getPrice();
      final int priceDiscounted = priceDiscount(price, customer);
      Optional<Account> account = customer.getAccounts().stream().filter(a->
(a.getBalance() > priceDiscounted)).findFirst();
      if (account.isPresent()) {
          order.setAccountId(account.get().getId());
          order.setStatus(OrderStatus.ACCEPTED);
          order.setPrice(priceDiscounted);
```

```
    } else {
        order.setStatus(OrderStatus.REJECTED);
    }
    return repository.add(order);
}

@PutMapping("/{id}")
public Order accept(@PathVariable Long id) {
    final Order order = repository.findById(id);
    template.put("http://account-service/withdraw/{id}/{amount}", null,
order.getAccountId(), order.getPrice());
    order.setStatus(OrderStatus.DONE);
    repository.update(order);
    return order;
}
// ...
}
```

这里比较有趣并值得注意的是，customer-service 的 GET /withAccounts/{id}方法（由 order-service 调用）也使用 Ribbon 客户端与另一个微服务 account-service 进行通信。以下是来自 CustomerController 的片段，它具有上述方法的实现。

```
@GetMapping("/withAccounts/{id}")
public Customer findByIdWithAccounts(@PathVariable("id") Long id) {
 Account[] accounts =
template.getForObject("http://account-service/customer/{customerId}",
Account[].class, id);
 Customer c = repository.findById(id);
 c.setAccounts(Arrays.stream(accounts).collect(Collectors.toList()));
 return c;
}
```

要测试该示例，可以按以下步骤操作。首先，使用 Maven 命令 mvn clean install 构建整个项目。然后，使用 java -jar 命令以任意顺序启动所有微服务，而无须任何其他参数。当然，开发人员也可以从集成开发环境中运行该应用程序。在启动时应该为每个微服务准备测试数据。由于没有持久性存储，因而重启后将删除所有对象。开发人员可以通过调用 order-service 公开的 POST 方法来测试整个系统。示例请求如下所示。

```
$ curl -d '{"productIds": [1,5],"customerId": 1,"status": "NEW"}' -H
"Content-Type: application/json" -X POST http://localhost:8090
```

如果尝试发送此请求，开发人员将能够看到 Ribbon 客户端打印的以下日志。

```
DynamicServerListLoadBalancer for client customer-service initialized:
DynamicServerListLoadBalancer:{NFLoadBalancer:name=customer-service,
current list of Servers=[localhost:8092],Load balancer stats=Zone
stats:{unknown=[Zone:unknown; Instance count:1; Active connections
count: 0;Circuit breaker tripped count: 0; Active connections per
server: 0.0;]},Server stats: [[Server:localhost:8092; Zone:UNKNOWN;
Total Requests:0; Successive connection failure:0; Total blackout
seconds:0; Last connection made:Thu Jan 01 01:00:00 CET 1970; First
connection made: Thu Jan 01 01:00:00 CET 1970; Active Connections:0;
total failure count in last (1000) msecs:0; average resp time:0.0;
90 percentile resp time:0.0; 95 percentile resp time:0.0; min resp
time:0.0; max resp time:0.0; stddev resp time:0.0]]} ServerList:
com.netflix.loadbalancer. ConfigurationBasedServerList@7f1e23f6
```

需要指出的是，本节描述的方法有一个很大的缺点，这使得它在由若干个微服务组成的系统中并不能很好地工作。如果系统有自动扩展功能的话，该问题会更严重。开发人员可以很容易地发现，所有服务的网络地址都是人工管理的。当然，我们也可以考虑将配置设置从每个胖 JAR 中的 application.yml 文件移动到配置服务器。但是，这样做并不能改变管理大量交互仍然很麻烦的事实。通过客户端负载均衡器和服务发现进行交互的能力，可以轻松解决这样的问题。

## 6.4　将 RestTemplate 与服务发现结合使用

实际上，与服务发现的集成是 Ribbon 客户端的默认行为。细心的读者可能还记得，我们可以通过设置 ribbon.eureka.enabled 属性为 false 禁用 Eureka 作为客户端均衡器的功能。在本节示例中开发人员将看到，服务发现的存在简化了对于服务之间通信的 Spring Cloud 组件的配置任务。

本示例的系统架构与前一个示例相同。要查看当前练习的源代码，必须切换到 ribbon_with_discovery 分支（https://github.com/piomin/shown here-spring-cloud-comm/tree/ribbon_with_discovery）。在这里，开发人员将看到的第一件事是新模块，即 discovery-service（发现服务）。本书第 4 章“服务发现”详细讨论了与 Eureka 相关的几乎所有方面，因而启动它时不应该有任何问题。我们需要运行一个具有真正基本设置的独立 Eureka 服务器，它在默认端口 8761 上可用。

与前面的示例相比，我们应该删除与 Ribbon 客户端严格相关的所有配置和注解。取

而代之的是，Eureka 发现客户端必须使用@EnableDiscoveryClient 启用，并且必须在 application.yml 文件中提供 Eureka 服务器地址。现在，order-service 的 main 类看起来应该如下所示。

```
@SpringBootApplication
@EnableDiscoveryClient
public class OrderApplication {
 @LoadBalanced
 @Bean
 RestTemplate restTemplate() {
 return new RestTemplate();
 }

 public static void main(String[] args) {
     new
SpringApplicationBuilder(OrderApplication.class).web(true).run(args);
 }
 // ...
}
```

以下是当前的配置文件。可以使用 spring.application.name 属性来设置服务的名称。

```
spring:
 application:
   name: order-service

server:
 port: ${PORT:8090}

eureka:
 client:
   serviceUrl:
     defaultZone: ${EUREKA_URL:http://localhost:8761/eureka/}
```

这里的设置和以前是一样的。我们还启动了所有的微服务。但是，这一次 account-service 服务和 product-service 服务的实例数将乘以 2。启动每个服务的第二个实例时，可以使用-DPORT 或-Dserver.port 参数覆盖默认服务器端口，如 java -jar -DPORT=9093 product-service-1.0-SNAPSHOT.jar。所有实例都已在 Eureka 服务器中注册，这可以使用其 UI 仪表板轻松查看，如图 6.2 所示。

Instances currently registered with Eureka

| Application | AMIs | Availability Zones | Status |
|---|---|---|---|
| ACCOUNT-SERVICE | n/a (2) | (2) | UP (2) - minkowp-l.p4.org:account-service:8091 , minkowp-l.p4.org:account-service:9091 |
| CUSTOMER-SERVICE | n/a (1) | (1) | UP (1) - minkowp-l.p4.org:customer-service:8092 |
| ORDER-SERVICE | n/a (1) | (1) | UP (1) - minkowp-l.p4.org:order-service:8090 |
| PRODUCT-SERVICE | n/a (2) | (2) | UP (2) - minkowp-l.p4.org:product-service:8093 , minkowp-l.p4.org:product-service:9093 |

图 6.2　查看已在 Eureka 服务器中注册的实例

这是本书第一次看到负载均衡的实际示例。默认情况下，Ribbon 客户端将在微服务的所有已注册实例之间平均分配流量。该算法称为轮询调度（Round Robin）。实际上，这意味着客户端会记住它转发的最后一个请求的位置，然后将当前请求发送到该行中的下一个服务。这个方法可能会被本书第 7 章所介绍的其他规则覆盖。通过在 ribbon.listOfServers 中设置以逗号分隔的服务地址列表，也可以为没有服务发现机制的前一个示例配置负载平衡，如 ribbon.listOfServers=localhost:8093,localhost:9093。回到本示例应用程序，order-service 服务发送的请求将在 account-service 服务和 product-service 服务的两个实例之间进行负载均衡。这和 customer-service 服务是相似的，只不过后者是在 account-service 服务的两个实例之间分配流量。如果开发人员已经启动了如图 6.2 所示的 Eureka 仪表板上的所有服务实例，并将一些测试请求发送到 order-service 服务，则肯定会看到以下日志。在该日志中，我们以加粗形式突出显示了某些片段，它们是 Ribbon 客户端显示的为目标服务找到的地址列表。

```
DynamicServerListLoadBalancer for client account-service initialized:
DynamicServerListLoadBalancer:{NFLoadBalancer:name=account-service,
current list of Servers=[minkowp-l.p4.org:8091, minkowp-l.p4.org:9091],
Load balancer stats=Zone stats: {defaultzone=[Zone:defaultzone; Instance
count:2; Active connections count: 0; Circuit breaker tripped count: 0;
Active connections per server: 0.0;]
 },Server stats: [[Server:minkowp-l.p4.org:8091; Zone:defaultZone; Total
Requests:0; Successive connection failure:0; Total blackout seconds:0;
Last connection made:Thu Jan 01 01:00:00 CET 1970; First connection made:
Thu Jan 01 01:00:00 CET 1970; Active Connections:0; total failure
count in last (1000) msecs:0; average resp time:0.0; 90 percentile
resp time:0.0; 95 percentile resp time:0.0; min resp time:0.0; max
resp time:0.0; stddev resp time:0.0],[Server:minkowp-l.p4.org:9091;
Zone:defaultZone; Total Requests:0; Successive connection failure:0;
Total blackout seconds:0; Last connection made:Thu Jan 01 01:00:00 CET
1970; First connection made: Thu Jan 01 01:00:00 CET 1970; Active
Connections:0; total failure count in last (1000) msecs:0; average
```

```
resp time:0.0; 90 percentile resp time:0.0; 95 percentile resp time:0.0;
min resp time:0.0; max resp time:0.0; stddev resp time:0.0]]}
ServerList:org.springframework.cloud.netflix.ribbon.eureka.
DomainExtractingServerList@3e878e67
```

## 6.5 使用 Feign 客户端

RestTemplate 是一个 Spring 组件，专门用于与 Spring Cloud 和微服务进行交互。但是，Netflix 开发了自己的工具，充当 Web 服务客户端，用于在独立 REST 服务之间提供现成可用的通信。其中的 Feign 客户端通常与带有@LoadBalanced 注解的 RestTemplate 作用相同，但是工作方式更加从容。它是一个 Java 到 HTTP 客户端绑定器，通过将注解处理为模板化请求来工作。使用 Open Feign 客户端时，开发人员只需创建一个接口并对其进行注解。它与 Ribbon 和 Eureka 集成，提供负载均衡的 HTTP 客户端，从服务发现中获取所有必需的网络地址。Spring Cloud 增加了对 Spring MVC 注解的支持，并使用了与 Spring Web 相同的 HTTP 消息转换器。

### 6.5.1 对不同区域的支持

现在回到上一个示例，我们将对它提出一些更改，以使其系统架构变得稍微复杂一些。如图 6.3 所示是当前架构的示意图。可以看到，微服务之间的通信模型仍然相同，但现在我们将对每个微服务启动两个实例并将它们分成两个不同的区域。在本书第 4 章“服务发现”中讨论使用 Eureka 的服务发现时，已经详细介绍了分区机制，所以我们假定开发人员已经熟悉该内容。本练习的主要目的不仅是演示如何使用 Feign 客户端，还将说明分区机制如何作用于微服务实例之间的通信。

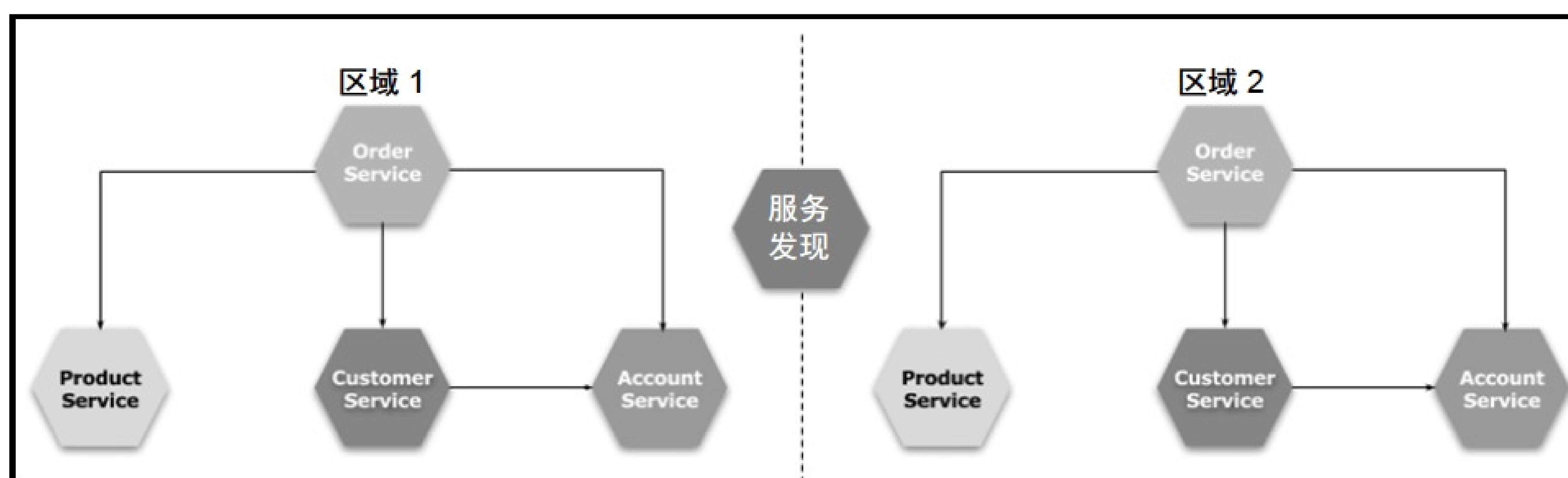



图 6.3　当前示例的架构

## 6.5.2　为应用程序启用 Feign

要在项目中包含 Feign，开发人员必须添加 spring-cloud-starter-feign 工件的依赖项或添加 Spring Cloud Netflix（版本至少需要为 1.4.0）的 spring-cloud-starter-openfeign。

```
<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-starter-feign</artifactId>
</dependency>
```

下一步是通过使用@EnableFeignClients 注解 main 类或配置类来为应用程序启用 Feign。此注解将导致搜索应用程序中实现的所有客户端。开发人员还可以通过设置 client 或 basePackages 注解属性来减少使用的客户端数量，如@EnableFeignClients (clients={AccountClient.class,Product.class})。以下是 order-service 服务的应用程序的 main 类。

```
@SpringBootApplication
@EnableDiscoveryClient
@EnableFeignClients
public class OrderApplication {

    public static void main(String[] args) {
        new
SpringApplicationBuilder(OrderApplication.class).web(true).run(args);
    }

    @Bean
    OrderRepository repository() {
        return new OrderRepository();
    }

}
```

### 1．构建 Feign 接口

有一种提供组件的方法只需要创建带有一些注解的接口即可，这也是 Spring Framework 的标准。对于 Feign，接口必须使用@FeignClient(name ="...")进行注解。它有一个必需的属性名称，如果启用了服务发现，则该属性名称对应于调用的微服务名称。否则，它与 url 属性一起使用，在 url 中可以设置具体的网络地址。

在这里，@FeignClient 并不是需要使用的唯一注解。在我们的客户端界面中，每个

方法都通过使用@RequestMapping 标记它来与特定的 HTTP API 端点相关联，当然也可以使用更具体的注解，如@GetMapping、@PostMapping 或@PutMapping，如本示例源代码片段中所示。

```
@FeignClient(name = "account-service")
public interface AccountClient {
    @PutMapping("/withdraw/{accountId}/{amount}")
    Account withdraw(@PathVariable("accountId") Long id,
@PathVariable("amount") int amount);
}

@FeignClient(name = "customer-service")
public interface CustomerClient {
    @GetMapping("/withAccounts/{customerId}")
    Customer findByIdWithAccounts(@PathVariable("customerId") Long
customerId);
}

@FeignClient(name = "product-service")
public interface ProductClient {
    @PostMapping("/ids")
    List<Product> findByIds(List<Long> ids);
}
```

这些组件可以注入控制器 bean，因为它们也是 Spring Beans。然后，开发人员只需要调用其方法。以下是 order-service 服务中 REST 控制器的当前实现。

```
@Autowired
OrderRepository repository;
@Autowired
AccountClient accountClient;
@Autowired
CustomerClient customerClient;
@Autowired
ProductClient productClient;

@PostMapping
public Order prepare(@RequestBody Order order) {
    int price = 0;
    List<Product> products =
productClient.findByIds(order.getProductIds());
    Customer customer =
```

```
customerClient.findByIdWithAccounts(order.getCustomerId());
    for (Product product : products)
        price += product.getPrice();
    final int priceDiscounted = priceDiscount(price, customer);
    Optional<Account> account = customer.getAccounts().stream().filter(a ->
(a.getBalance() > priceDiscounted)).findFirst();
    if (account.isPresent()) {
        order.setAccountId(account.get().getId());
        order.setStatus(OrderStatus.ACCEPTED);
        order.setPrice(priceDiscounted);
    } else {
        order.setStatus(OrderStatus.REJECTED);
    }
    return repository.add(order);
}
```

### 2. 启动微服务

我们已经更改了 application.yml 中所有微服务的配置。现在，有两个不同的配置文件，第一个用于将应用程序分配给 zone1，第二个用于 zone2。开发人员可以查看 feign_with_discovery 分支的版本（https://github.com/piomin/shown here-spring-cloud-comm/tree/feign_with_discovery）。然后，使用 mvn clean install 命令构建整个项目。应该使用 java -jar --spring.profiles.active=zone[n]命令启动应用程序，其中，[n]是区域的编号。

因为必须启动许多实例来执行该测试，所以通过设置-Xmx 参数（如-Xmx128m）来考虑对堆大小的限制是值得的。以下是其中一个微服务的当前配置设置。

```
spring:
 application:
     name: account-service

---
spring:
 profiles: zone1
eureka:
 instance:
     metadataMap:
         zone: zone1
client:
    serviceUrl:
        defaultZone: http://localhost:8761/eureka/
```

```
        preferSameZoneEureka: true
server:
 port: ${PORT:8091}

---
spring:
 profiles: zone2
eureka:
 instance:
     metadataMap:
        zone: zone2
 client:
     serviceUrl:
        defaultZone: http://localhost:8761/eureka/
        preferSameZoneEureka: true
server:
 port: ${PORT:9091}
```

我们将为每个区域的每个微服务启动一个实例。因此，会有 9 个正在运行的 Spring Boot 应用程序（包括服务发现服务器），如图 6.4 所示。

Instances currently registered with Eureka

| Application | AMIs | Availability Zones | Status |
| --- | --- | --- | --- |
| ACCOUNT-SERVICE | n/a (2) | (2) | UP (2) - minkowp-l.p4.org:account-service:8091 , minkowp-l.p4.org:account-service:9091 |
| CUSTOMER-SERVICE | n/a (2) | (2) | UP (2) - minkowp-l.p4.org:customer-service:9092 , minkowp-l.p4.org:customer-service:8092 |
| ORDER-SERVICE | n/a (2) | (2) | UP (2) - minkowp-l.p4.org:order-service:9090 , minkowp-l.p4.org:order-service:8090 |
| PRODUCT-SERVICE | n/a (2) | (2) | UP (2) - minkowp-l.p4.org:product-service:8093 , minkowp-l.p4.org:product-service:9093 |

图 6.4　每个区域的每个微服务都启动了一个实例

如果将测试请求发送到在 zone1（http://localhost:8090）中运行的 order-service 服务实例，则所有流量将转发到该区域中的其他服务，而对于 zone2（http://localhost:9090）也是一样的。以下以粗体突出显示的是 Ribbon 客户端打印的在当前区域中注册的目标服务的已找到地址的列表片段。

```
DynamicServerListLoadBalancer for client product-service initialized:
DynamicServerListLoadBalancer:{NFLoadBalancer:name=product-service,current
list of Servers=[minkowp-l.p4.org:8093],Load balancer stats=Zone stats:
{zone1=[Zone:zone1; Instance count:1; Active connections count: 0; Circuit
breaker tripped count: 0; Active connections per server: 0.0;]...
```

### 6.5.3　继承支持

开发人员可能已经注意到，控制器实现中的注解和该控制器所服务的 REST 服务的 Feign 客户端实现是相同的。我们可以创建一个包含抽象 REST 方法定义的接口。该接口可以由控制器类实现，也可以由 Feign 客户端接口扩展。

```
public interface AccountService {

    @PostMapping
    Account add(@RequestBody Account account);

    @PutMapping
    Account update(@RequestBody Account account);

    @PutMapping("/withdraw/{id}/{amount}")
    Account withdraw(@PathVariable("id") Long id, @PathVariable("amount")
int amount);

    @GetMapping("/{id}")
    Account findById(@PathVariable("id") Long id);

    @GetMapping("/customer/{customerId}")
    List<Account> findByCustomerId(@PathVariable("customerId") Long
customerId);

    @PostMapping("/ids")
    List<Account> find(@RequestBody List<Long> ids);

    @DeleteMapping("/{id}")
    void delete(@PathVariable("id") Long id);

}
```

现在，控制器类为基本接口中的所有方法提供了一个实现，但不包含任何 REST 映射的注解。当然，@RestController 注解还是需要的。以下是 account-service 服务控制器的一个片段。

```
@RestController
public class AccountController implements AccountService {

    @Autowired
```

```
    AccountRepository repository;

    public Account add(@RequestBody Account account) {
        return repository.add(account);
    }
    // ...
}
```

用于调用 account-service 服务的 Feign 客户端接口不提供任何方法。它只是扩展了基础接口 AccountService。要查看基于接口和 Feign 继承的完整实现，可以切换到以下 feign_with_inheritance 分支。

```
https://github.com/piomin/shown here-spring-cloud-comm/tree/feign_with_
inheritance
```

以下是一个带继承支持的 Feign 客户端声明示例。它扩展了 AccountService 接口，并处理了由@RestController 公开的所有方法。

```
@FeignClient(name = "account-service")
public interface AccountClient extends AccountService {
}
```

### 6.5.4　手动创建客户端

如果开发人员不太相信像注解这样的形式，则可以始终使用 Feign Builder API 手动创建 Feign 客户端。Feign 有若干个可以自定义的功能，如消息的编码器和解码器或 HTTP 客户端实现。

```
AccountClient accountClient = Feign.builder().client(new OkHttpClient())
    .encoder(new JAXBEncoder())
    .decoder(new JAXBDecoder())
    .contract(new JAXRSContract())
    .requestInterceptor(new BasicAuthRequestInterceptor("user",
"password"))
    .target(AccountClient.class, "http://account-service");
```

### 6.5.5　客户端的自定义

客户端的自定义不仅可以使用 Feign Builder API 接口执行，还可以使用类似注解的形式来执行。例如，开发人员可以通过使用@FeignClient 注解的 configuration 属性来设置

它，以提供配置类。

```
@FeignClient(name = "account-service", configuration =
AccountConfiguration.class)
```

以下显示了一个示例配置 bean。

```
@Configuration
public class AccountConfiguration {
 @Bean
 public Contract feignContract() {
     return new JAXRSContract();
 }

 @Bean
 public Encoder feignEncoder() {
     return new JAXBEncoder();
 }

 @Bean
 public Decoder feignDecoder() {
     return new JAXBDecoder();
 }

 @Bean
 public BasicAuthRequestInterceptor basicAuthRequestInterceptor() {
     return new BasicAuthRequestInterceptor("user", "password");
 }
}
```

Spring Cloud 支持通过声明 Spring Beans 覆盖以下属性。

- Decoder：默认为 ResponseEntityDecoder。
- Encoder：默认为 SpringEncoder。
- Logger：默认为 Slf4jLogger。
- Contract：默认为 SpringMvcContract。
- Feign.Builder：默认为 HystrixFeign.Builder。
- Client：如果启用了 Ribbon，则为 LoadBalancerFeignClient；否则，将使用默认的 Feign 客户端。
- Logger.Level：它为 Feign 设置默认日志级别。可以在 NONE、BASIC、HEADERS 和 FULL 之间选择。
- Retryer：它允许在通信失败的情况下实现重试算法。

- ❑ ErrorDecoder：它允许将 HTTP 状态代码映射到特定于应用程序的异常。
- ❑ Request.Options：它允许为请求设置读取和连接超时。
- ❑ Collection<RequestInterceptor>：已注册的 RequestInterceptor 实现的集合，这些实现将基于从请求中获取的数据执行某些操作。

也可以使用配置属性自定义 Feign 客户端。例如，可以通过在 feign.client.config 属性前缀之后提供客户端的名称来覆盖所有可用客户端的设置，或者仅覆盖单个选定客户端的设置。如果将名称设置为 default 而不是特定的客户端名称，则会将其应用于所有 Feign 客户端。使用@EnableFeignClients 注解及其 defaultConfiguration 属性时，也可以使用与前面所述类似的方式指定默认配置。appplication.yml 文件中提供的设置始终具有比@Configuration bean 更高的优先级。要更改该方法，并且如果开发人员更喜欢使用@Configuration 注解而不是 YAML 文件，则应该将 feign.client.default-to-properties 属性设置为 false。以下是 account-service 服务的 Feign 客户端配置示例，它设置了连接超时、HTTP 连接的读取超时和日志级别。

```
feign:
 client:
   config:
     account-service:
       connectTimeout: 5000
       readTimeout: 5000
       loggerLevel: basic
```

## 6.6 小　结

本章启动了若干个可以相互通信的微服务。我们讨论了诸如 REST 客户端的不同实现、多个实例之间的负载均衡以及与服务发现的集成等主题。在我们看来，这些方面非常重要，所以需要在两个章节中对它们进行更详细地说明。本章的重点是介绍微服务之间的通信，并且讨论了与微服务架构的其他重要组件的集成。第 7 章将演示负载均衡器和 REST 客户端的更高级用法，特别关注网络和通信问题。阅读完本章之后，开发人员应该能够在应用程序中正确使用 Ribbon、Feign，甚至 RestTemplate，并将它们连接到其他 Spring Cloud 组件。

在大多数情况下，有这些知识就已经足够了。但是，有时开发人员需要自定义客户端负载均衡器配置或启用更高级的通信机制，如断路器或回退逻辑。了解这些解决方案及其对系统中的服务间通信的影响非常重要。我们将在第 7 章讨论它们。

# 第 7 章　高级负载均衡和断路器

本章将继续讨论第 6 章所涉猎的主题，即服务间通信。我们将把该主题扩展到更高级的负载均衡、超时和断路示例。

Spring Cloud 提供的功能使得微服务之间的通信实现既简单又便捷。但是，不应忘记的是，我们在这种通信中遇到的主要困难均涉及系统的处理时间。如果开发人员的系统中有许多微服务，那么需要处理的首要问题之一就是延迟（Delay）问题。本章将讨论一些 Spring Cloud 功能，这些功能可以帮助开发人员避免延迟问题。这些延迟问题是在处理单个输入请求时，由于服务之间的跳数（Hop）太多，来自多个服务的响应缓慢或服务暂时不可用而导致的。有若干种策略都可以处理部分失败，这些策略包括设置网络超时、限制等待请求的数量、实现不同的负载均衡方法，或者设置断路器模式和回退实现等。

我们还将再次讨论 Ribbon 和 Feign 客户端，这次将重点关注其更高级的配置功能。本章将介绍一个全新的库，即 Netflix Hystrix。该库实现了断路器模式。

本章将要讨论的主题包括：

- 使用 Ribbon 客户端的不同负载均衡算法。
- 为应用程序启用断路器。
- 使用配置属性自定义 Hystrix。
- 使用 Hystrix 仪表板监控服务间通信。
- 联合使用 Hystrix 和 Feign 客户端。

## 7.1　负载均衡规则

Spring Cloud Netflix 提供了不同的负载均衡算法，以便为用户提供不同的帮助。具体选择哪一种方法取决于开发人员的需求。在 Netflix OSS 术语中，该算法称为规则（Rule）。自定义规则类应该已经实现了 IRule 基本接口。Spring Cloud 中默认可用以下实现。

- RoundRobinRule：此规则将简单地使用众所周知的轮询调度算法选择服务器，在该算法中，传入的请求将按顺序分布在所有实例上。它通常用作更高级规则的默认规则或回退逻辑，此类规则有 ClientConfigEnabledRoundRobinRule 和 ZoneAvoidanceRule 等。例如，ZoneAvoidanceRule 就是 Ribbon 客户端的默认规则。

- AvailabilityFilteringRule：此规则将跳过标记为电路跳闸（Circuit Tripped）或具有大量并发连接的服务器。它还将使用 RoundRobinRule 作为基类。默认情况下，如果 HTTP 客户端连续 3 次无法与其建立连接，则实例会电路跳闸。可以使用 niws.loadbalancer.<clientName>.connectionFailureCountThreshold 属性自定义此方法。一旦实例电路跳闸，它将在下一次重试之前的下一个 30 秒内保持这种状态。也可以在配置设置中覆盖此属性。
- WeightedResponseTimeRule：通过此实现，实例的流量转发器（Traffic Volume Forwarder）与实例的平均响应时间成反比。换句话说，响应时间越长，它的权重就越小。在这种情况下，负载均衡客户端将记录服务的每个实例的流量和响应时间。
- BestAvailableRule：根据类说明文档中的描述，此规则会跳过具有跳闸断路器的服务器，并选择具有最低并发请求的服务器。

**注意：**

跳闸断路器（Tripped Circuit Breaker）是一个取自电气工程的术语，它意味着没有电流流过电路。在 IT 术语中，它指的是发送到服务的连续请求太多失败的情况，因此，客户端上的软件会立即中断调用远程服务的任何进一步尝试，以便释放服务器端的应用程序。

### 7.1.1　WeightedResponseTime 规则

到目前为止，我们通常都是通过从 Web 浏览器或 REST 客户端调用服务来手动测试它们。但是，当前的更改不允许这样的方法，因为我们需要为服务设置假延迟，并且还要生成许多 HTTP 请求。

### 7.1.2　引入 Hoverfly 进行测试

在这一点上，我们想引入一个有趣的框架，它可能是这类测试的完美解决方案。这个框架就是 Hoverfly，它是一种用于存根（Stub）或模拟 HTTP 服务的轻量级服务虚拟化工具。它最初是用 Go 编写的，但也为开发人员提供了一个用于管理 Java 中的 Hoverfly 的富有表现力的 API。Hoverfly Java 由 SpectoLabs 维护，它提供了抽象二进制和 API 调用的类、用于创建模拟的领域专用语言（Domain Specified Language，DSL），以及与 JUnit 测试框架的集成。这个框架有一个笔者个人非常喜欢的功能。开发人员可以通过调用 DSL 定义中的一种方法，轻松地为每个模拟服务添加延迟。要为项目启用 Hoverfly，开发人员必须在 Maven 的 pom.xml 中包含以下依赖项。

```
<dependency>
    <groupId>io.specto</groupId>
    <artifactId>hoverfly-java</artifactId>
    <version>0.9.0</version>
    <scope>test</scope>
</dependency>
```

### 7.1.3　测试规则

本节所讨论的示例可以在 GitHub 上找到。要访问它，必须切换到 weighted_lb 分支（https://github.com/piomin/sample-spring-cloud-comm/tree/weighted _lb）。JUnit 测试类名为 CustomerControllerTest，它位于 src/test/java 目录下。要启用 Hoverfly 测试，应该定义 JUnit @ClassRule。HoverflyRule 类提供了一个 API，该 API 允许开发人员使用不同的地址、特征（Characteristics）和响应来模拟许多服务。在下面的源代码片段中，你可能会看到我们的示例微服务 account-service 的两个实例已经在@ClassRule 中声明。你可能还记得，该服务已经由 customer-service 服务和 order-service 服务调用。

现在来看一看 customer-service 服务模块中的测试类。它使用端口 8091 和 9091 上可用的 account-service 服务的两个实例的预定义响应来模拟 GET /customer/*方法。第一个延迟了 200 毫秒，而第二个则延迟了 50 毫秒。

```
@ClassRule
public static HoverflyRule hoverflyRule = HoverflyRule
 .inSimulationMode(dsl(
 service("account-service:8091")
     .andDelay(200, TimeUnit.MILLISECONDS).forAll()
     .get(startsWith("/customer/"))
.willReturn(success("[{\"id\":\"1\",\"number\":\"1234567890\",
\"balance\":5 000}]", "application/json")),
 service("account-service:9091")
     .andDelay(50, TimeUnit.MILLISECONDS).forAll()
     .get(startsWith("/customer/"))
.willReturn(success("[{\"id\":\"2\",\"number\":\"1234567891\",
\"balance\":8 000}]", "application/json"))))
 .printSimulationData();
```

在运行该测试之前，还应该修改 ribbon.listOfServers 配置文件，方法是将其更改为 listOfServers: account-service:8091，account-service:9091。只应该在使用 Hoverfly 时进行这样的修改。

以下是一个用于测试用例的 test 方法，它将调用由 customer-service 服务公开的 GET

/withAccounts/ {id}端点 1000 次。在这 1000 次中，每一次都将使用该客户所拥有的账户列表，依次调用 account-service 服务的 GET customer/{customerId}端点。每一次请求都将使用 WeightedResponseTimeRule 规则在 account-service 服务的两个实例之间进行负载均衡。

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = WebEnvironment.DEFINED_PORT)
public class CustomerControllerTest {

    private static Logger LOGGER =
LoggerFactory.getLogger(CustomerControllerTest.class);

    @Autowired
    TestRestTemplate template;
    // ...

    @Test
    public void testCustomerWithAccounts() {
        for (int i = 0; i < 1000; i++) {
            Customer c = template.getForObject("/withAccounts/{id}",
Customer.class, 1);
            LOGGER.info("Customer: {}", c);
        }
    }

}
```

使用加权响应规则（Weighted Response Rule）实现的方法非常有趣。在开始测试之后，传入的请求在两个 account-service 服务实例之间以 50:50 的比例进行负载均衡。但是，经过一段时间后，大多数都会转发给具有更小延迟的实例。

以笔者个人在本地计算机上启动的 JUnit 测试为例，最终，端口 9091 上的实例处理了 731 个请求，端口 8091 上的实例处理了 269 个请求。但是，在测试的末尾阶段，该比例看起来有点不同，并且加权更有利于具有较小延迟的实例，其中传入流量在两个实例之间的比例大致被划分为 4:1。

现在，可以通过添加第三个 account-service 服务实例来稍微改变一下我们的测试用例，新服务实例的延迟大约为 10 秒。此修改旨在模拟 HTTP 通信中的超时。以下是来自 JUnit 的@ClassRule 定义的片段，其中最新的服务实例将侦听端口 10091。

```
service("account-service:10091")
    .andDelay(10000, TimeUnit.MILLISECONDS).forAll()
```

```
    .get(startsWith("/customer/"))
.willReturn(success("[{\"id\":\"3\",\"number\":\"1234567892\",
\"balance\":1 0000}]", "application/json"))
```

相应地，我们还应该在 Ribbon 配置中执行以下更改，以便为最新的 account-service 服务实例启用负载均衡。

```
listOfServers: account-service:8091, account-service:9091, account-
service:10091
```

还必须执行的最后一项修改，就是 RestTemplate bean 声明。虽然和上一个测试用例一样，它需要获得保留，但是在本实例中，开发人员可以将读取和连接超时都设置为 1 秒，因为在测试期间启动的第三个 account-service 服务实例的延迟为 10 秒，所以发送到那里的每个请求都会在 1 秒后因为超时而终止。

```
@LoadBalanced
@Bean
RestTemplate restTemplate(RestTemplateBuilder restTemplateBuilder) {
    return restTemplateBuilder
        .setConnectTimeout(1000)
        .setReadTimeout(1000)
        .build();
}
```

如果此时运行与以前相同的测试，其结果将不会令人满意。所有声明的实例之间的分配将是：420 个请求由侦听端口 8091（延迟 200 毫秒）的实例处理，468 个请求由侦听端口 9091（延迟 50 毫秒）的实例处理，并且仍然还有 112 个请求被发送到第三个实例，它们显然会由于超时而被终止。为什么会出现这样的统计数据呢？我们可以将默认的负载均衡规则从 WeightedResponseTimeRule 更改为 AvailabilityFilteringRule，然后再重新运行测试。经过这样的修改之后，现在将会向第一个和第二个实例各发送 496 个请求，而只有 8 个请求被发送到第三个实例，并且一秒超时。有趣的是，如果将默认规则设置为 BestAvailableRule，则所有请求都将发送到第一个实例。

通过上述示例，相信开发人员已经完全明白了 Ribbon 客户端的所有可用负载均衡规则之间的差异。

## 7.2　自定义 Ribbon 客户端

可以使用 Spring bean 声明覆盖 Ribbon 客户端的多个配置设置。与 Feign 一样，它应

该在名为 configuration 的客户端注解字段中声明，如@RibbonClient(name= "account- service", configuration=RibbonConfiguration.class)。使用此方法可以自定义以下功能。

- IClientConfig：其默认实现是 DefaultClientConfigImpl。
- IRule：此组件将用于确定应从列表中选择哪一个服务实例。值得一提的是，ZoneAvoidanceRule 实现类是自动配置的。
- IPing：这是一个在后台运行的组件。它负责确保服务实例正在运行。
- ServerList<Server>：这可以是静态的也可以是动态的。如果它是动态的（由 DynamicServerListLoadBalancer 使用），那么后台线程将以预定义的间隔刷新并过滤列表。默认情况下，Ribbon 使用从配置文件中获取的静态服务器列表。它由 ConfigurationBasedServerList 实现。
- ServerListFilter<Server>：ServerListFilter 同样是一个组件，它将由 DynamicServerListLoadBalancer 使用，可以过滤从 ServerList 实现返回的服务器。该接口有两种实现，即自动配置的 ZonePreferenceServerListFilter 和 ServerListSubsetFilter。
- ILoadBalancer：它负责在客户端的服务的可用实例之间执行负载均衡。默认情况下，Ribbon 将使用 ZoneAwareLoadBalancer。
- ServerListUpdater：它负责更新给定应用程序的可用实例列表。默认情况下，Ribbon 将使用 PollingServerListUpdater。

现在让我们来看一个配置类示例，它将定义 IRule 和 IPing 组件的默认实现。通过提供 @RibbonClients(defaultConfiguration=RibbonConfiguration.class) 注解，可以为单个 Ribbon 客户端以及应用程序类路径中可用的所有 Ribbon 客户端定义如下所示的配置。

```
@Configuration
public class RibbonConfiguration {

    @Bean
    public IRule ribbonRule() {
        return new WeightedResponseTimeRule();
    }

    @Bean
    public IPing ribbonPing() {
        return new PingUrl();
    }

}
```

即使没有使用 Spring 的经验，开发人员可能也已经猜到（基于之前的示例），这里也可以使用 properties 文件自定义配置。在这种情况下，Spring Cloud Netflix 将与由 Netflix 提供的 Ribbon 说明文档中描述的属性兼容。以下这些类就是受到支持的属性，它们应该以<clientName>.ribbon 为前缀，或者如果它们适用于所有客户端，则仅使用 ribbon 作为前缀。

- NFLoadBalancerClassName：ILoadBalancer 默认实现类。
- NFLoadBalancerRuleClassName：IRule 默认实现类。
- NFLoadBalancerPingClassName：IPing 默认实现类。
- NIWSServerListClassName：ServerList 默认实现类。
- NIWSServerListFilterClassName：ServerListFilter 默认实现类。

以下示例与前面的@Configuration 类相似，它将覆盖 Spring Cloud 应用程序使用的 IRule 和 IPing 默认实现。

```
account-service:
 ribbon:
   NFLoadBalancerPingClassName: com.netflix.loadbalancer.PingUrl
   NFLoadBalancerRuleClassName:
com.netflix.loadbalancer.WeightedResponseTimeRule
```

## 7.3　带 Hystrix 的断路器模式

前文已经讨论过 Spring Cloud Netflix 中负载均衡器算法的不同实现。其中一些基于监视实例响应时间或失败次数。在这些情况下，负载均衡器会根据这些统计信息决定应该调用哪一个实例。断路器模式（Circuit Breaker Pattern）应该被视为该解决方案的扩展。断路器背后的主要思想非常简单，受保护的函数调用将包含在断路器对象中，该对象负责监视故障调用的数量。如果故障达到阈值，则电路断开，所有其他调用将自动失败。通常情况下，如果断路器跳闸，则开发人员会希望有某种监控警报。在应用程序中使用断路器模式所带来的一些重要好处是：应用程序能够在相关服务发生故障时继续运行，防止出现级联故障，并且能够给予服务恢复的时间。

### 7.3.1　使用 Hystrix 构建应用程序

Netflix 在其库中提供了一个名为 Hystrix 的断路器模式实现。该库也被包含在 Spring Cloud 的断路器的默认实现中。Hystrix 还有一些其他有趣的功能，也应该被视为处理分布

式系统的延迟和容错的综合工具。重要的是，如果断路器被断开，则 Hystrix 会将所有调用重定向到指定的回退方法（Fallback Method）。回退方法旨在提供通用响应，而不需要依赖网络（一般来说，会从内存缓存中读取响应或仅实现为静态逻辑）。如果因为某些原因必须执行网络调用，则建议使用其他的 HystrixCommand 或 HystrixObservableCommand 实现它。要在项目中包含 Hystrix，则应该为 Spring Cloud Netflix（请注意，必须是早于 1.4.0 的版本）使用 spring-cloud-starter-netflix-hystrix 或 spring-cloud-starter-hystrix 启动器。

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-hystrix</artifactId>
</dependency>
```

### 1. 实现 Hystrix 的命令

Spring Cloud Netflix Hystrix 将查找使用@HystrixCommand 进行注解的方法，然后将其包裹在连接到断路器的代理对象（Proxy Object）中。由于这个原因，Hystrix 能够监控这种方法的所有调用。此注解目前仅适用于标有@Component 或@Service 的类。这对于我们来说是很重要的信息，因为我们已经在 REST 控制器类中实现了与所有先前示例中调用的其他服务相关的逻辑，该控制器类采用的标记是@RestController 注解。因此，在 customer-service 服务应用程序中，所有逻辑都已移至新创建的 CustomerService 类，然后该类将被注入控制器 bean。负责与 account-service 服务通信的方法已使用@HystrixCommand 进行了注解。此外，我们还实现了一个回退方法，其名称将传递到 fallbackMethod 注解的字段中。此方法仅返回一个空列表。

```
@Service
public class CustomerService {

    @Autowired
    RestTemplate template;
    @Autowired
    CustomerRepository repository;

    // ...

    @HystrixCommand(fallbackMethod = "findCustomerAccountsFallback")
    public List<Account> findCustomerAccounts(Long id) {
        Account[] accounts =
template.getForObject("http://account-service/customer/{customerId}",
Account[].class, id);
```

```
        return Arrays.stream(accounts).collect(Collectors.toList());
    }

    public List<Account> findCustomerAccountsFallback(Long id) {
        return new ArrayList<>();
    }

}
```

不要忘记用@EnableHystrix 标记 main 类，因为它会告诉 Spring Cloud，应该为应用程序使用断路器。当然，也可以选择使用@EnableCircuitBreaker 来注解一个类，它的作用是一样的。出于测试目的，account-service.ribbon.listOfServers 属性应该包含 localhost:8091 和 localhost:9091 服务的两个实例的网络地址。

虽然我们已经为 Ribbon 客户端声明了 account-service 服务的两个实例，但是在这里我们将仅启动 8091 端口上可用的实例。如果调用 customer-service 服务方法 GET http://localhost:8092/withAccounts/{id}，则 Ribbon 将尝试对这两个已声明的实例之间的每个传入请求进行负载均衡，也就是说，一旦接收到一个包含账户列表的响应，那么第二次将收到一个空账户列表，反之亦然。这在下面的应用程序日志片段中可以看得很清楚。要访问该示例应用程序的源代码，开发人员需要切换到与第 6 章中的示例相同的 GitHub 存储库的 hystrix_basic 分支（https://github.com/piomin/sample-spring-cloud-comm/tree/hystrix_basic）。

```
{"id":1,"name":"John Scott","type":"NEW","accounts":[]}

{"id":1,"name":"John
Scott","type":"NEW","accounts":[{"id":1,"number":"1234567890",
"balance":5000},{"id":2,"number":"1234567891","balance":5000},
{"id":3,"number":"12345678 92","balance":0}]}
```

### 2. 使用缓存数据实现回退

上一个示例中提供的回退实现非常简单。返回空列表对于在生产模式中运行的应用程序没有什么实际意义。反之，在应用程序中使用回退方法则更有意义，例如，当请求失败时，可以从缓存中读取数据。像这样的缓存可以在客户端应用程序内部实现，或者也可以使用第三方工具实现，如 Redis、Hazelcast 或 EhCache。在 Spring Framework 中也提供了最简单的实现，并且在将 spring-boot-starter-cache 工件与依赖项一起包含之后即可使用。要为 Spring Boot 应用程序启用缓存，应该使用@EnableCaching 注解 main 或配置类，并在以下上下文环境中提供 CacheManager bean。

```
@SpringBootApplication
@RibbonClient(name = "account-service")
@EnableHystrix
@EnableCaching
public class CustomerApplication {

    @LoadBalanced
    @Bean
    RestTemplate restTemplate() {
        return new RestTemplate();
    }

    public static void main(String[] args) {
        new
SpringApplicationBuilder(CustomerApplication.class).web(true).run(args);
    }

    @Bean
    public CacheManager cacheManager() {
        return new ConcurrentMapCacheManager("accounts");
    }
    // ...

}
```

然后，可以使用@CachePut 注解来标记被断路器包裹（Wrap）的方法，这将把从调用方法返回的结果添加到缓存映射（Cache Map）。在这种情况下，该映射被命名为 accounts。最后，可以通过直接调用 CacheManager bean 来读取回退方法实现中的数据。如果多次重试相同的请求，则开发人员将看到空的账户列表不再作为响应返回。相反，该服务将始终返回在第一次成功调用期间缓存的数据。

```
@Autowired
CacheManager cacheManager;
@CachePut("accounts")
@HystrixCommand(fallbackMethod = "findCustomerAccountsFallback")
public List<Account> findCustomerAccounts(Long id) {
    Account[] accounts =
template.getForObject("http://account-service/customer/{customerId}",
Account[].class, id);
    return Arrays.stream(accounts).collect(Collectors.toList());
}
```

```
public List<Account> findCustomerAccountsFallback(Long id) {
    ValueWrapper w = cacheManager.getCache("accounts").get(id);
    if (w != null) {
        return (List<Account>) w.get();
    } else {
        return new ArrayList<>();
    }
}
```

### 7.3.2　跳闸断路器

现在不妨来做一个练习。截至目前，开发人员已经学习了如何使用 Hystrix 与 Spring Cloud 一起在应用程序中启用和实现断路器，以及如何使用回退方法从缓存中获取数据。但是，仍然没有使用跳闸断路器来防止负载均衡器调用故障的实例。现在，如果故障百分比大于 30%，则我们想要配置 Hystrix 在 3 次失败的调用尝试之后断开电路，并防止在接下来的 5 秒内调用 API 方法。测量时间窗口约为 10 秒。为了满足这些要求，开发人员必须覆盖若干个默认的 Hystrix 配置设置。它可以使用@HystrixCommand 中的@HystrixProperty 注解来执行。

以下是负责从 customer-service 服务获取账户列表的方法的当前实现。

```
@CachePut("accounts")
@HystrixCommand(fallbackMethod = "findCustomerAccountsFallback",
 commandProperties = {
    @HystrixProperty(name =
"execution.isolation.thread.timeoutInMilliseconds", value = "500"),
    @HystrixProperty(name = "circuitBreaker.requestVolumeThreshold",
value= "10"),
    @HystrixProperty(name = "circuitBreaker.errorThresholdPercentage",
value = "30"),
    @HystrixProperty(name = "circuitBreaker.sleepWindowInMilliseconds",
value = "5000"),
    @HystrixProperty(name = "metrics.rollingStats.timeInMilliseconds",
value = "10000")
 }
)
public List<Account> findCustomerAccounts(Long id) {
    Account[] accounts =
template.getForObject("http://account-service/customer/{customerId}",
Account[].class, id);
    return Arrays.stream(accounts).collect(Collectors.toList());
}
```

有关 Hystrix 配置属性的完整列表，请访问 Netflix 的 GitHub 站点（https://github.com/Netflix/Hystrix/wiki/Configuration）。本章不会详细讨论所有这些属性，而只是关注对于微服务之间的通信来说最为重要的属性。以下是我们的示例中使用过的属性的列表及其说明。

- execution.isolation.thread.timeoutInMilliseconds：此属性将设置以毫秒为单位的时间，在此时间之后将发生读取或连接超时，客户端将终止命令执行。Hystrix 将这种方法调用标记为失败，并执行回退逻辑。通过将 command.timeout.enabled 属性设置为 false，可以完全关闭该超时设置。默认值为 1000 毫秒。
- circuitBreaker.requestVolumeThreshold：此属性可以设置滚动窗口中将使电路跳闸的最小请求数。默认值为 20。在我们的示例中，此属性被设置为 10，这意味着前 9 次申请即使全部失败也不会使电路跳闸。请注意，在设置了该值之后，因为前面已经假设如果 30%的传入请求失败，就应该断开电路，所以最小的传入请求数其实是 3。
- circuitBreaker.errorThresholdPercentage：此属性可以设置最小错误百分比，超过此百分比会导致断开电路，系统开始短路请求回退逻辑。默认值为 50。在上面的示例中已经将其设置为 30，因为我们希望在有 30%的请求失败的情况下即断开电路。
- circuitBreaker.sleepWindowInMilliseconds：此属性设置跳闸电路和允许尝试以确定是否应再次断开电路之间的一段时间。在此期间，所有传入的请求都将被拒绝。默认值为 5000（毫秒）。因为我们想在电路断开后第一次重新尝试调用之前等待 10 秒，所以将其设置为 10000。
- metrics.rollingStats.timeInMilliseconds：此属性可以设置统计滚动窗口的持续时间（以毫秒为单位）。这是 Hystrix 为断路器的使用和发布而保留的指标（Metrics）时间。

通过这些设置，开发人员可以运行与前一个示例相同的 JUnit 测试。我们将使用 HoverflyRule 启动两个 account-service 服务存根。其中第一个将延迟 200 毫秒，而第二个将延迟 2000 毫秒，该值大于使用 execution.isolation.thread.timeoutInMilliseconds 属性为 @HystrixCommand 设置的超时（默认为 1000 毫秒）。

在运行 JUnit CustomerControllerTest 后，即可查看打印的日志。以下就是从笔者的机器上启动的测试中获取的日志。可以看到，第一个请求来自 account-service 服务，被负载均衡到第一个实例，延迟了 200 毫秒。为方便查看，笔者在日志后面以双斜杠注释方式

给它添加了标记(1)，后面的标记格式相同，兹不赘述。

接下来，发送到 9091 端口上可用实例的每个请求因为需延迟 2000 毫秒，所以都会在一秒钟后因为超时而结束。在发送 10 次请求之后，首次由于故障而导致电路跳闸，详见日志标记(2)。

然后，在接下来的 10 秒内，每个请求都由一个回退方法处理，该方法将返回缓存的数据，见日志标记(3)和(4)。

10 秒之后，客户端尝试再次调用 account-service 服务实例，因为它再次被负载均衡到延迟为 200 毫秒，所以它成功了，见日志标记(5)。

这种成功导致电路的闭合。但糟糕的是，接下来的 account-service 服务实例仍然响应缓慢（因为又要延迟 2000 毫秒），因而上述情况会再次发生，直到 JUnit 测试结束，见日志标记(6)和(7)。

以上就是关于 Hystrix 的断路器在 Spring Cloud 中的工作方式的详细说明。

```
16:54:04+01:00 Found response delay setting for this request host:
{account-service:8091 200} // (1)
16:54:05+01:00 Found response delay setting for this request host:
{account-service:9091 2000}
16:54:05+01:00 Found response delay setting for this request host:
{account-service:8091 200}
16:54:06+01:00 Found response delay setting for this request host:
{account-service:9091 2000}
16:54:06+01:00 Found response delay setting for this request host:
{account-service:8091 200}
...
16:54:09+01:00 Found response delay setting for this request host:
{account-service:9091 2000} // (2)
16:54:10.137 Customer [id=1, name=John Scott, type=NEW, accounts=[Account
[id=1, number=1234567890, balance=5000]]] // (3)
...
16:54:20.169 Customer [id=1, name=John Scott, type=NEW, accounts=[Account
[id=1, number=1234567890, balance=5000]]] // (4)
16:54:20+01:00 Found response delay setting for this request host:
{account-service:8091 200} // (5)
16:54:20+01:00 Found response delay setting for this request host:
{account-service:9091 2000}
16:54:21+01:00 Found response delay setting for this request host:
{account-service:8091 200}
...
16:54:25+01:00 Found response delay setting for this request host:
```

```
{account-service:8091 200} // (6)
16:54:26.157 Customer [id=1, name=John Scott, type=NEW, accounts=[Account
[id=1, number=1234567890, balance=5000]]] // (7)
```

## 7.4　监控延迟和容错

如前文所述，Hystrix 并不是一个仅能实现断路器模式的简单工具。它是一种可以处理分布式系统中延迟和容错的解决方案。Hystrix 提供的一个有趣功能是能够公开与服务间通信相关的最重要指标，并通过用户界面仪表板显示它们。此功能适用于使用 Hystrix 命令包裹的客户端。

在之前的一些示例中，我们仅分析了系统的一部分，以模拟 customer-service 服务和 account-service 服务之间的通信延迟。在测试高级负载均衡算法或不同的断路器配置设置时，这是一个非常好的方法，但现在我们将回过头来分析为一组独立的 Spring Boot 应用程序设置的作为一个一体化的示例系统，这将使得开发人员能够观察 Spring Cloud 与 Netflix OSS 工具一起工作的方式，了解它们如何帮助监控和响应微服务之间的通信中的延迟和失败问题。该示例系统将以简单的方式模拟故障。它具有静态配置，其中包括 product-service 服务和 account-service 服务的两个实例的网络地址，但每个服务只运行其中一个实例。

为了加深印象并区别于上一个示例，本示例的系统架构考虑了有关故障的假设，其示意图如图 7.1 所示。

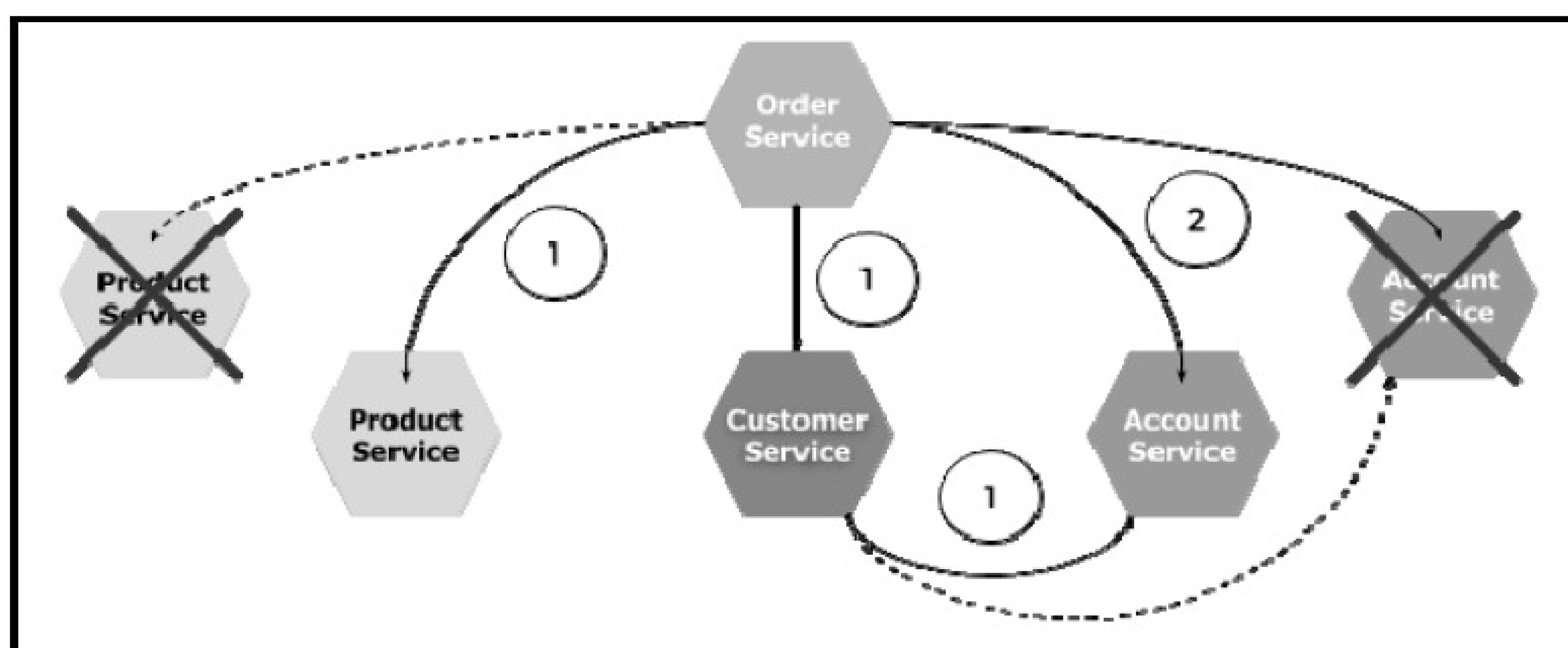



图 7.1　本示例的系统架构

这一次的测试也有所不同。以下是测试方法的片段，它在循环中被调用。首先，它

将调用 order-service 服务的 POST http://localhost:8090/端点，发送 Order 对象，并接收具有 id、status 和 price 集合的响应。该请求在图 7.1 中被标记为①，order-service 服务与 product-service 服务和 customer-service 服务通信，此外，customer-service 服务将调用 account-service 服务的端点。如果订单已被接收，则测试客户端将使用订单的 id 调用 PUT http://localhost:8090/{id}方法接收该订单并从该账户中提取资金。在服务器端，这种情况下只有一个服务间通信，在图 7.1 中标记为②。在运行此测试之前，必须启动属于系统一部分的所有微服务。

```
Random r = new Random();
Order order = new Order();
order.setCustomerId((long) r.nextInt(3)+1);
order.setProductIds(Arrays.asList(new Long[] {(long) r.nextInt(10)+1,(long)
r.nextInt(10)+1}));
order = template.postForObject("http://localhost:8090", order,
Order.class); // 对应图 7.1 中的①
if (order.getStatus() != OrderStatus.REJECTED) {
    template.put("http://localhost:8090/{id}", null, order.getId()); // 对
应图 7.1 中的②
}
```

### 7.4.1　公开 Hystrix 的指标流

使用 Hystrix 与其他微服务进行通信的每个微服务都可能公开使用 Hystrix 命令包裹的每个集成的指标。要启用此类指标流（Metrics Stream），应该在 spring-boot-starter-actuator 上包含依赖项，这会将/hystrix.stream 对象公开为管理端点。此外，开发人员还必须包含 spring-cloud-starter-hystrix，因为它已经被添加到我们的示例应用程序中。

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

已经生成的流将进一步公开为 JSON 条目，这些 JSON 条目中将包含表示方法中的单一调用的特征的指标。以下是来自 customer-service 服务的 GET /withAccounts/{id}方法中的单个条目。

```
{"type":"HystrixCommand","name":"customer-service.findWithAccounts",
"group":"CustomerService","currentTime":1513089204882,
```

```
"isCircuitBreakerOpen":false,"errorPercentage":0,"errorCount":0,
"requestCount":74,"rollingCountBadRequests":0,
"rollingCountCollapsedRequests":0,"rollingCountEmit":0,
"rollingCountExceptionsThrown":0,"rollingCountFailure":0,
"rollingCountFallbackEmit":0,"rollingCountFallbackFailure":0,
"rollingCountFallbackMissing":0,"rollingCountFallbackRejection":0,
"rollingCountFallbackSuccess":0,"rollingCountResponsesFromCache":0,
"rollingCountSemaphoreRejected":0,"rollingCountShortCircuited":0,
"rollingCountSuccess":75,"rollingCountThreadPoolRejected":0,
"rollingCountTimeout":0,"currentConcurrentExecutionCount":0,
"rollingMaxConcurrentExecutionCount":1,"latencyExecute mean":5,
"latencyExecute":{"0":0,"25":0,"50":0,"75":15,"90":16,"95":31,
"99":47,"99.5":47,"100":62},"latencyTotal mean":5,
"latencyTotal":{"0":0,"25":0,"50":0,"75":15,"90":16,"95":31,
"99":47,"99.5":47,"100":62},
"propertyValue circuitBreakerRequestVolumeThreshold":10,
"propertyValue circuitBreakerSleepWindowInMilliseconds":10000,
"propertyValue circuitBreakerErrorThresholdPercentage":30,
"propertyValue circuitBreakerForceOpen":false,
"propertyValue circuitBreakerForceClosed":false,
"propertyValue circuitBreakerEnabled":true,
"propertyValue executionIsolationStrategy":"THREAD",
"propertyValue executionIsolationThreadTimeoutInMilliseconds":2000,
"propertyValue executionTimeoutInMilliseconds":2000,
"propertyValue executionIsolationThreadInterruptOnTimeout":true,
"propertyValue executionIsolationThreadPoolKeyOverride":null,
"propertyValue executionIsolationSemaphoreMaxConcurrentRequests":10,
"propertyValue_fallbackIsolationSemaphoreMaxConcurrentRequests":10,
"propertyValue_metricsRollingStatisticalWindowInMilliseconds":10000,
"propertyValue_requestCacheEnabled":true,
"propertyValue_requestLogEnabled":true,"reportingHosts":1,
"threadPool":"Custom erService"}
```


### 7.4.2　Hystrix 仪表板

Hystrix 仪表板将可视化以下信息。

- 运行状况和流量将显示为一个圆圈，这个圆圈会更改其颜色和大小，以反映传入的统计信息的更改。
- 过去 10 秒内的错误百分比。
- 按数字显示最后两分钟的请求率，在图表上显示结果。

- ❑ 断路器状态。断开为 Open，闭合为 Closed。
- ❑ 服务主机的数量。
- ❑ 最后一分钟的延迟百分位数。
- ❑ 服务的线程池。

### 1．使用仪表板构建应用程序

Hystrix 仪表板与 Spring Cloud 集成在一起。在系统内部实现仪表板时，最好的方法是将独立的 Spring Boot 应用程序与仪表板分开。要在项目中包含 Hystrix 仪表板，可以使用 spring-cloud-starter-hystrix-netflix-dashboard 启动器；对于 Spring Cloud Netflix 来说则是使用 spring-cloud-starter-hystrix-dashboard。请注意，Spring Cloud Netflix 版本必须早于 1.4.0。

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-hystrix-dashboard</artifactId>
</dependency>
```

应用程序的 main 类应该使用@EnableHystrixDashboard 进行注解。在启动之后，Hystrix 仪表板即在/hystrix 上下文路径下可用。

```
@SpringBootApplication
@EnableHystrixDashboard
public class HystrixApplication {

    public static void main(String[] args) {
        new
SpringApplicationBuilder(HystrixApplication.class).web(true).run(args);
    }

}
```

本示例系统已经将端口 9000 配置为 Hystrix 应用程序的默认设置，并且这是在 hystrix-dashboard 模块中实现的。因此，在启动 hystrix-dashboard 后，如果在 Web 浏览器中调用 http://localhost:9000/hystrix 地址，它将显示如图 7.2 所示的页面。在该页面，开发人员应该提供 Hystrix 流端点的地址，以及可选的标题。如果要显示从 order-service 服务调用的所有端点的指标，可输入地址 http://localhost:8090/hystrix.stream，然后单击 Monitor Steam（监控流）按钮。

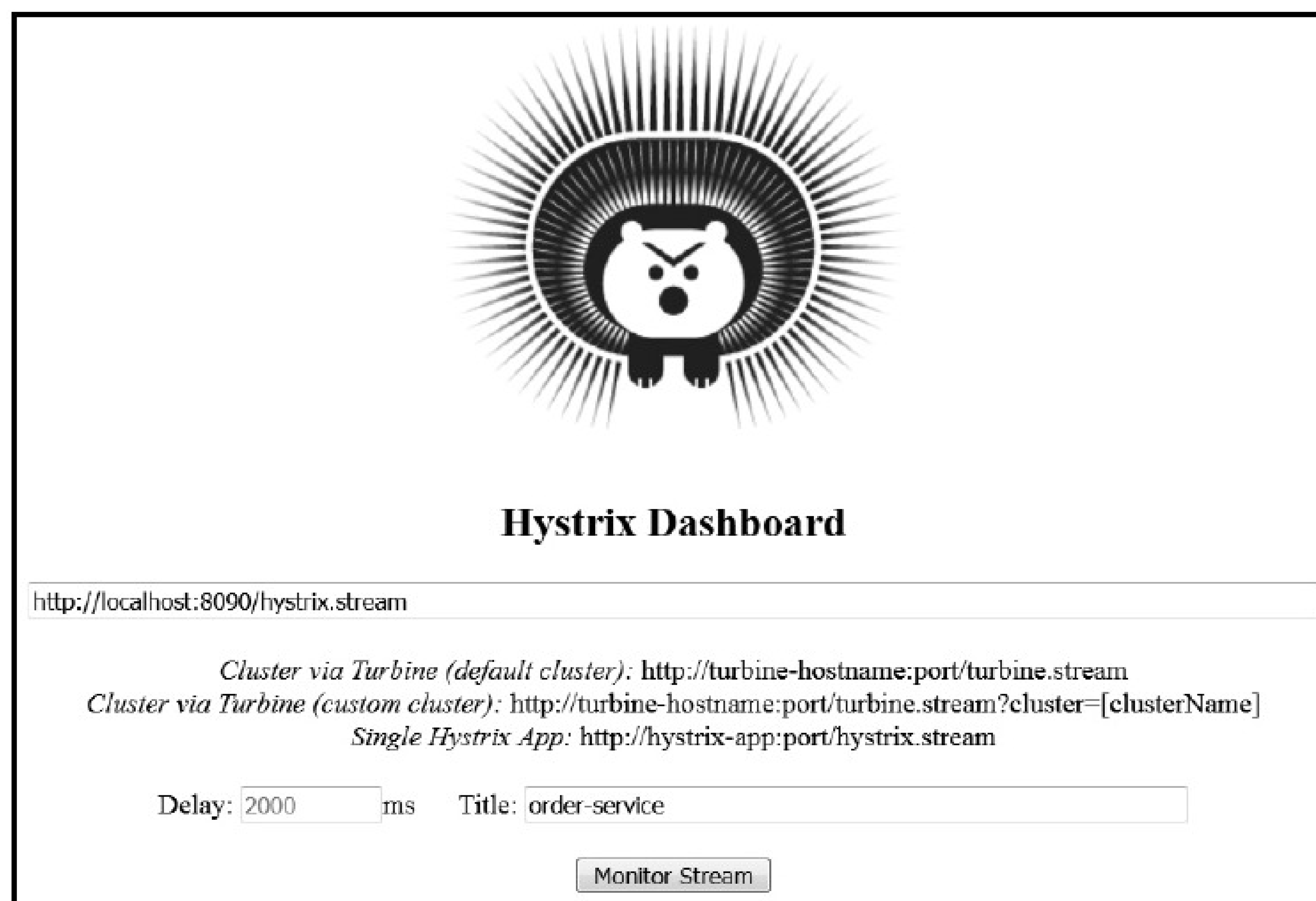



图 7.2　hystrix-dashboard 仪表板界面

### 2. 监控仪表板上的指标

现在我们将介绍如何调用 customer-service 服务的 GET /withAccounts/{id}方法。它被@HystrixCommand 包裹，显示在 Hystrix 仪表板上，并且在 customer-service.findWithAccounts 标题下，取自 commandKey 属性。此外，用户界面仪表板还将显示有关分配给每个 Spring Bean 的线程池的信息，这些线程池提供了使用 Hystrix 命令包裹方法的实现。在这种情况下，它是 CustomerService。

```
@Service
public class CustomerService {

    // ...
    @CachePut("customers")
    @HystrixCommand(commandKey = "customer-service.findWithAccounts",
fallbackMethod = "findCustomerWithAccountsFallback",
        commandProperties = {
            @HystrixProperty(name =
"execution.isolation.thread.timeoutInMilliseconds", value = "2000"),
            @HystrixProperty(name =
"circuitBreaker.requestVolumeThreshold", value = "10"),
            @HystrixProperty(name =
"circuitBreaker.errorThresholdPercentage", value = "30"),
            @HystrixProperty(name =
```

```
"circuitBreaker.sleepWindowInMilliseconds", value = "10000"),
            @HystrixProperty(name =
"metrics.rollingStats.timeInMilliseconds", value = "10000")
        })
    public Customer findCustomerWithAccounts(Long customerId) {
        Customer customer =
template.getForObject("http://customer-service/withAccounts/{id}",
Customer.class, customerId);
        return customer;
    }

    public Customer findCustomerWithAccountsFallback(Long customerId) {
        ValueWrapper w =
cacheManager.getCache("customers").get(customerId);
        if (w != null) {
            return (Customer) w.get();
        } else {
            return new Customer();
        }
    }

}
```

如图 7.3 所示是 JUnit 测试开始后 Hystrix 仪表板的屏幕截图。在该图中，我们监视的是使用@HystrixCommand 包裹的所有 3 个方法的状态。正如预期的那样，product-service 服务的 findByIds 方法的电路已经断开。几秒钟之后，account-service 服务的 withdraw 方法的电路也已经断开。

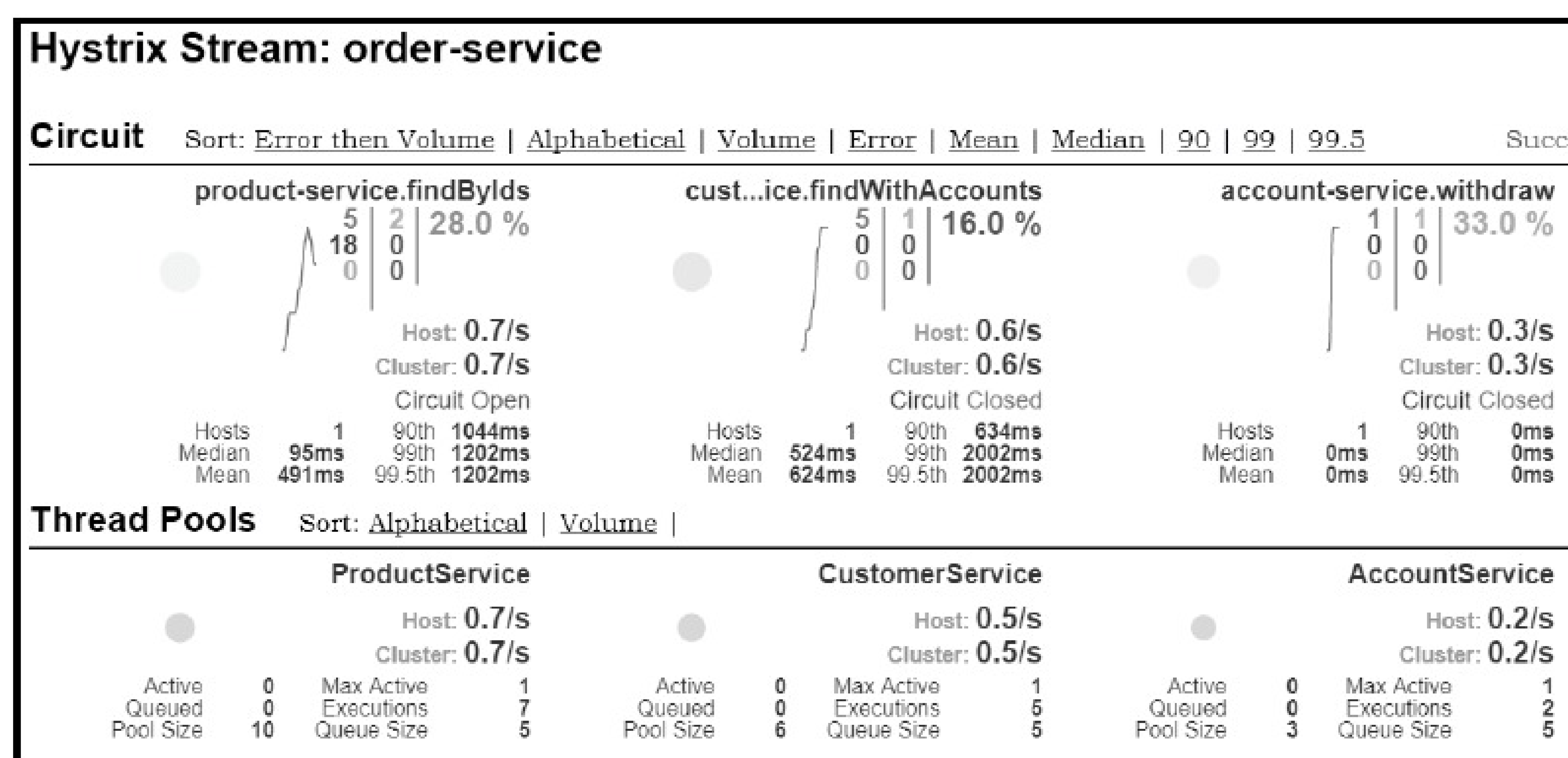



图 7.3　JUnit 测试开始后 Hystrix 仪表板的屏幕截图

片刻之后，情况将会稳定下来。所有电路都保持闭合状态，因为只有一小部分流量被发送到应用程序的非活动实例。这显示了 Spring Cloud 与 Hystrix 和 Ribbon 的强大功能。系统能够自动重新配置自身，以便根据负载均衡器和断路器生成的指标将大多数传入请求重定向到工作实例，如图 7.4 所示。

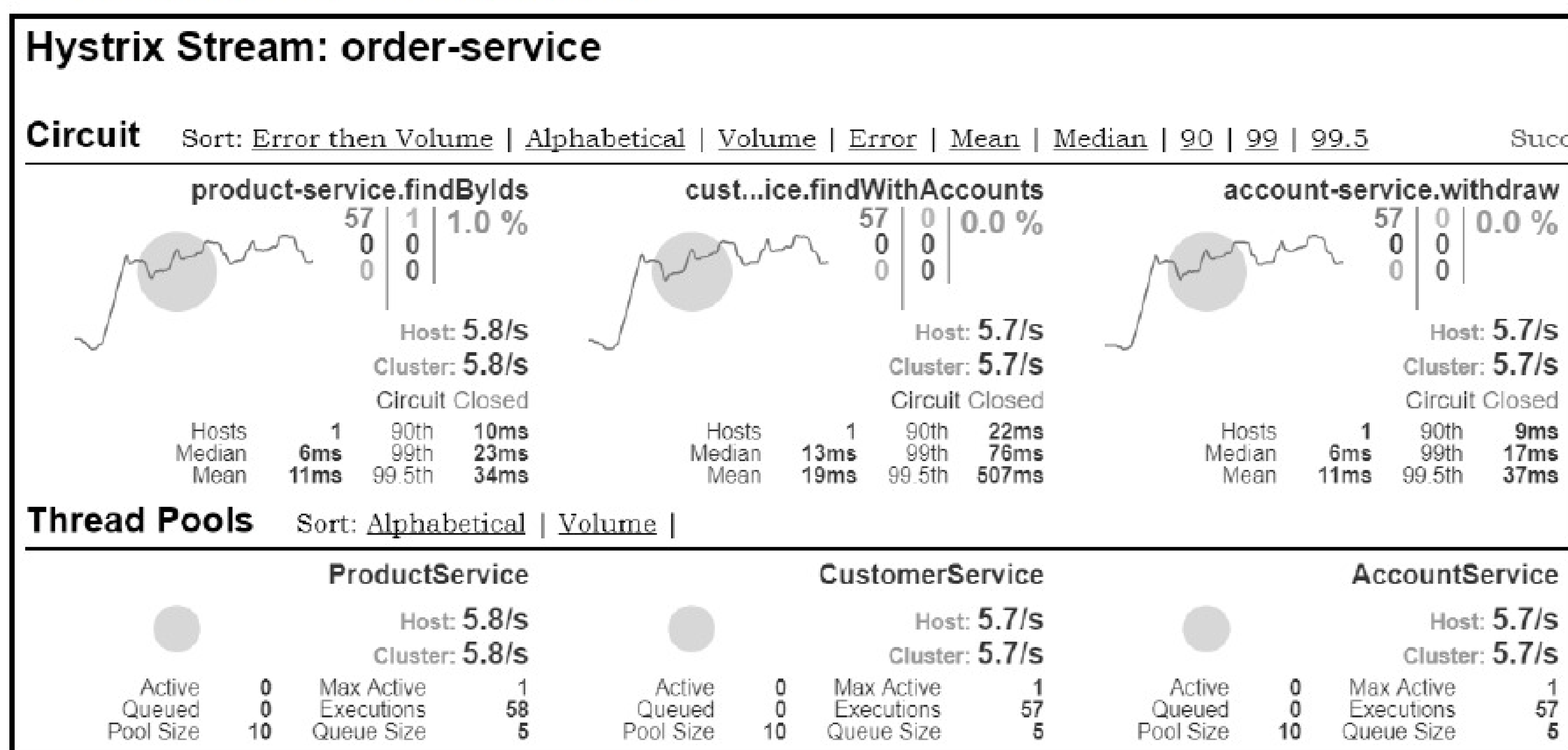



图 7.4　进入稳定工作状态的 Hystrix 仪表板显示

### 3．使用 Turbine 聚集 Hystrix 的流

开发人员可能已经注意到，我们只能在 Hystrix 仪表板中查看该服务的单个实例。当我们显示 order-service 服务的命令状态时，customer-service 服务和 account-service 服务之间的通信便没有指标数据，反之亦然。可以想象，如果 order-service 服务有多个实例在运行，那肯定会使我们手忙脚乱，因为必须在 Hystrix 仪表板中的不同实例或服务之间定期切换。幸运的是，有一个名为 Turbine 的应用程序，它可以将所有相关的/hystrix.stream 端点聚合到一个组合的/turbine.stream 中，使我们能够轻松监控整个系统的整体运行状况。

（1）启用 Turbine

在做出修改以便为示例应用程序启用 Turbine 之前，应该从启用服务发现开始，这是启用 Turbine 所必须的。切换到 hystrix_with_turbine 分支以访问支持使用 Eureka 进行服务发现并使用 Turbine 聚合 Hystrix 流的示例系统版本。要为公开用户界面仪表板的项目启用 Turbine，只需在依赖项中包含 spring-cloud-starter-turbine，并使用@EnableTurbine 注解应用程序的 main 类。

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
```

```
    <artifactId>spring-cloud-starter-turbine</artifactId>
</dependency>
```

turbine.appConfig 配置属性是 Turbine 将用于查找实例的 Eureka 服务名称的列表。然后，可以在 URL（http://localhost:9000/turbine.stream）下的 Hystrix 仪表板中使用 Turbine 流。该地址也可以由 turbine.aggregator.clusterConfig 属性的值确定，如 http://localhost:9000/turbine.stream?cluster=<clusterName>。如果该名称是 default，则可以省略 cluster 参数。以下是 Turbine 配置，它可以将所有 Hystrix 的可视化指标结合在一个用户界面仪表板中。

```
turbine:
 appConfig: order-service,customer-service
   clusterNameExpression: "'default'"
```

现在，整个示例系统的所有 Hystrix 指标都显示在一个仪表板站点中。我们需要显示的只是监视统计信息流，这可在 http://localhost:9000/turbine.stream 下找到，如图 7.5 所示。

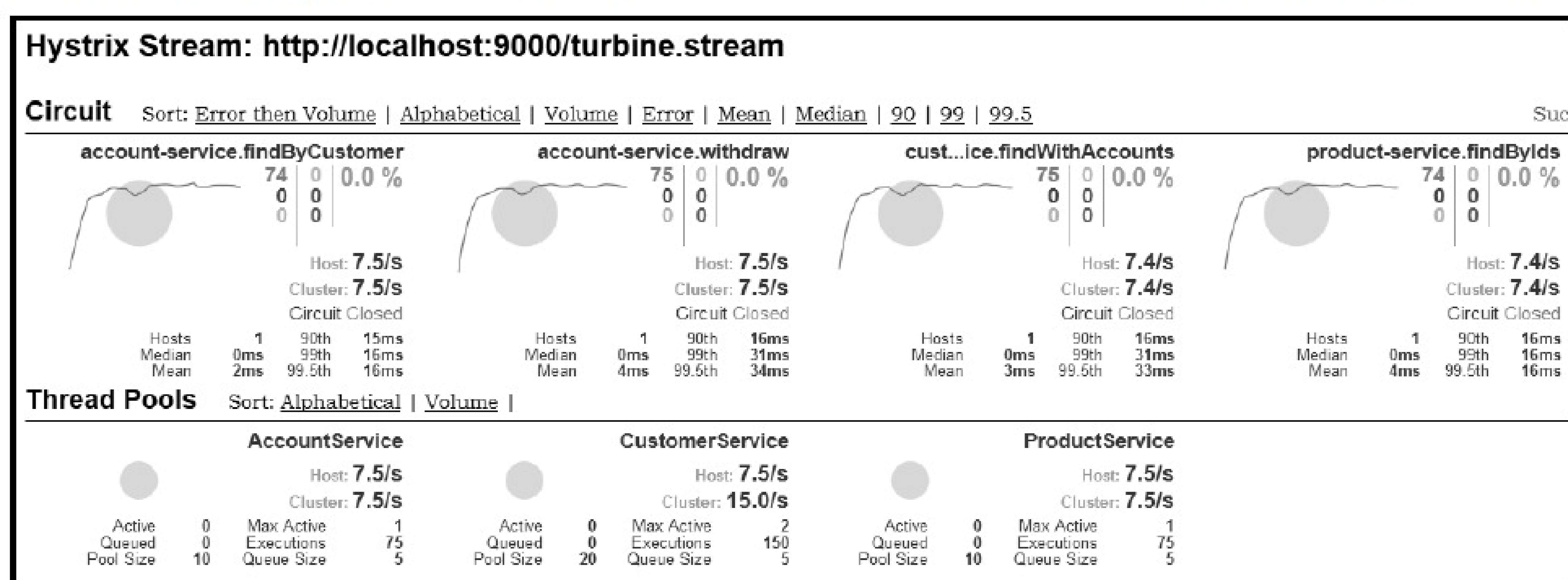



图 7.5　聚合所有 Hystrix 指标

开发人员也可以通过提供一个具有 turbine.aggregator.clusterConfig 属性的服务的列表来为每个服务配置一个集群。在这种情况下，开发人员可以通过提供具有 http://localhost:9000/turbine.stream?cluster=ORDER-SERVICE 参数的服务名 cluster 在集群之间切换。该集群的名称必须以大写形式提供，因为 Eureka 服务器返回的值是大写的。

```
turbine:
  aggregator:
    clusterConfig: ORDER-SERVICE,CUSTOMER-SERVICE
  appConfig: order-service,customer-service
```

默认情况下，Turbine 将在 Eureka 中已注册实例的 homePageUrl 地址下查找该实例的

/hystrix.stream 端点，然后它会将/hystrix.stream 附加到该 URL。本示例应用程序的 order-service 服务是在端口 8090 下启动的，因此，还应该将默认管理端口覆盖到 8090。order-service 服务的当前配置显示在以下代码片段中。或者，开发人员也可以使用 eureka.instance.metadata-map.management.port 属性更改该端口。

```
spring:
 application:
   name: order-service

server:
 port: ${PORT:8090}

eureka:
 client:
   serviceUrl:
     defaultZone: ${EUREKA URL:http://localhost:8761/eureka/}

management:
 security:
   enabled: false
     port: 8090
```

（2）通过流传输启用 Turbine

虽然经典 Turbine 模块可以从所有分布式 Hystrix 命令中提取指标，但这并不总是一个好选择。诸如从 HTTP 端点收集指标数据之类的操作也可以使用消息代理以异步方式实现。要通过流传输（Streaming）启用 Turbine，应该在项目中包含以下依赖项，然后再使用@EnableTurbineStream 注解主应用程序。虽然以下示例使用了 RabbitMQ 作为默认的消息代理，但是开发人员也可以通过包含 spring-cloud-starter-stream-kafka 来使用 Apache Kafka。

```
<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-starter-turbine-stream</artifactId>
</dependency>
<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-starter-stream-rabbit</artifactId>
</dependency>
```

上述代码中的依赖项应包含在服务器端。对于客户端应用程序来说，这些是 order-

service 服务和 customer-service 服务，开发人员需要添加 spring-cloud-netflix-hystrix-stream 库。如果已经在本地运行消息代理，则它应该已在自动配置的设置上成功运行。开发人员也可以使用 Docker 容器运行 RabbitMQ，就像我们在本书第 5 章“使用 Spring Cloud Config 进行分布式配置”的 Spring Cloud Config 集成使用 AMQP 的示例中所做的那样。然后，开发人员应该在 application.yml 中为客户端和服务器端应用程序覆盖以下属性。

```
spring:
 rabbitmq:
   host: 192.168.99.100
   port: 5672
   username: guest
   password: guest
```

如果登录到 http://192.168.99.100:15672 下的 RabbitMQ 管理控制台，开发人员将看到在本示例应用程序启动后创建了名为 springCloudHystrixStream 的新交换消息。现在，我们唯一要做的就是运行和上一小节的经典 Turbine 方法示例相同的 JUnit 测试，所有指标数据都通过消息代理发送，并且可以在 http://localhost:9000 端点下观察。如果开发人员想要自己尝试，可切换到 hystrix_with_turbine_stream 分支（有关详细信息，请参阅 https://github.com/piomin/sample-spring-cloud-comm/tree/hystrix_with_turbine_stream）。

## 7.5　故障和带有 Feign 的断路器模式

默认情况下，Feign 客户端与 Ribbon 和 Hystrix 集成在一起。这意味着，如果开发人员愿意的话，可以在使用该库时应用不同的方法来处理系统中的延迟和超时。第一种方法是 Ribbon 客户端提供的连接重试机制；第二种方法是断路器模式和 Hystrix 项目下可用的回退实现，这已在本章前面的章节中讨论过。

### 7.5.1　重试与 Ribbon 的连接

使用 Feign 库时，默认情况下会为应用程序启用 Hystrix。这意味着如果开发人员不想使用它，则应在配置设置中禁用它。为了使用 Ribbon 测试重试机制，建议禁用 Hystrix。为了启用 Feign 的连接重试，只需设置两个配置属性-MaxAutoRetries 和 MaxAutoRetries NextServer。在这种情况下，重要的设置也是 ReadTimeout 和 ConnectTimeout。所有这些都可以在 application.yml 文件中覆盖。以下是最重要的 Ribbon 设置列表。

- MaxAutoRetries：这是同一服务器或服务实例上的最大重试次数。第一次尝试被

排除在此计数之外。

- MaxAutoRetriesNextServer：这是要重试的下一个服务器或服务实例的最大数量，不包括第一个服务器。
- OkToRetryOnAllOperations：这表示可以为该客户端重试所有操作。
- ConnectTimeout：这是等待与服务器或服务实例建立连接的最长时间。
- ReadTimeout：这是建立连接后等待服务器响应的最长时间。

现在假设已经有两个目标服务的实例。与第一个服务实例的连接已经建立，但是响应速度太慢，发生超时。客户端将根据 MaxAutoRetries=1 属性对该服务实例执行一次重试。如果仍未成功，则尝试连接该服务的第二个可用实例。根据 MaxAutoRetriesNextServer = 2 属性中设置的内容，在出现失败时，此操作将重复两次。如果上述描述的机制最终不成功，则会因为超时而返回到外部客户端。在这种情况下，即使超过 4 秒钟的情况也可能发生。看看以下配置。

```
ribbon:
 eureka:
   enabled: true
 MaxAutoRetries: 1
 MaxAutoRetriesNextServer: 2
 ConnectTimeout: 500
 ReadTimeout: 1000

feign:
 hystrix:
   enabled: false
```

该解决方案是为基于微服务的环境实现的标准重试机制。开发人员还可以了解一些与 Ribbon 的超时和重试的不同配置设置相关的其他方案。所以需要将这种机制与 Hystrix 的断路器一起使用。但是，开发人员必须记住，ribbon.ReadTimeout 属性的值应该低于 Hystrix 的 execution.isolation.thread.timeoutInMilliseconds 属性的值。

建议开发人员测试上述配置设置，并把它作为一项练习。可以使用先前介绍的 Hoverfly JUnit 规则来模拟服务实例的延迟和存根。

## 7.5.2　Hystrix 对 Feign 的支持

如前文所述，在使用 Feign 库时，默认情况下会为应用程序启用 Hystrix，但这仅适用于旧版本的 Spring Cloud。根据最新版 Spring Cloud 的说明文档，开发人员应该将 feign.hystrix.enabled 属性设置为 true，这会强制 Feign 用断路器包裹所有方法。

**注意：**

在 Spring Cloud Dalston 发布之前，如果 Hystrix 在类路径上，则 Feign 默认会将所有方法包裹在断路器中。

在 Spring Cloud Dalston 版本中更改了此默认行为，转而采用了选择性加入的方法。

将 Hystrix 与 Feign 客户端一起使用时，要提供以前在@HystrixCommand 中使用@HystrixProperty 设置的配置属性，最简单的方法是通过 application.yml 文件。以下是之前提供的示例的等效配置。

```
hystrix:
 command:
   default:
     circuitBreaker:
       requestVolumeThreshold: 10
       errorThresholdPercentage: 30
       sleepWindowInMilliseconds: 10000
     execution:
       isolation:
         thread:
           timeoutInMilliseconds: 1000
     metrics:
       rollingStats:
         timeInMilliseconds: 10000
```

Feign 支持回退表示法。要为给定的@FeignClient 启用回退机制，应该使用提供回退实现的类名设置 fallback 属性。该实现类应该定义为 Spring Bean。

```
@FeignClient(name = "customer-service", fallback =
CustomerClientFallback.class)
public interface CustomerClient {

    @CachePut("customers")
    @GetMapping("/withAccounts/{customerId}")
    Customer findByIdWithAccounts(@PathVariable("customerId") Long
customerId);

}
```

回退实现基于缓存，并且将实现使用@FeignClient 注解的接口。

```
@Component
public class CustomerClientFallback implements CustomerClient {
```

```
    @Autowired
    CacheManager cacheManager;

    @Override
    public Customer findByIdWithAccountsFallback(Long customerId) {
        ValueWrapper w =
cacheManager.getCache("customers").get(customerId);
        if (w != null) {
            return (Customer) w.get();
        } else {
            return new Customer();
        }
    }

}
```

或者，也可以考虑实现 FallbackFactory 类。这种方法有一个很大的优势，它使开发人员可以访问导致回退触发的原因。要为 Feign 声明 FallbackFactory 类，只需使用@FeignClient 中的 fallbackFactory 属性。

```
@FeignClient(name = "account-service", fallbackFactory =
AccountClientFallbackFactory.class)
public interface AccountClient {

    @CachePut
    @GetMapping("/customer/{customerId}")
    List<Account> findByCustomer(@PathVariable("customerId") Long
customerId);

}
```

自定义 FallbackFactory 类需要实现一个 FallbackFactory 接口，该接口将声明一个必须被覆盖的 T create（Throwable cause）方法。

```
@Component
public class AccountClientFallbackFactory implements
FallbackFactory<AccountClient> {

    @Autowired
    CacheManager cacheManager;

    @Override
```

```
    public AccountClient create(Throwable cause) {
        return new AccountClient() {
            @Override
            List<Account> findByCustomer(Long customerId) {
                ValueWrapper w =
cacheManager.getCache("accounts").get(customerId);
                if (w != null) {
                    return (List<Account>) w.get();
                } else {
                    return new Customer();
                }
            }
        }
    }
}
```

## 7.6　小　　结

如果开发人员常使用自动配置的客户端进行服务间通信，则可能不了解本章中介绍的配置设置或工具。但是，我们认为开发人员应该对这些高级机制有所了解，即使它们可以在后台运行或有现成可用的东西。本章尝试通过一些简单示例演示了它们的工作原理，使开发人员可以更详细地了解诸如负载均衡器、重试、回退或断路器之类的主题。阅读本章后，开发人员应该能够自定义 Ribbon、Hystrix 或 Feign 客户端，以满足与微服务之间的通信相关的需求，无论是小规模还是大规模需求。开发人员还应该了解在系统中使用它们的时间和原因。到本章为止，基于微服务架构中核心元素的讨论就到此结束了。

接下来，我们将把目光放到系统之外，讨论一个更重要的组件：网关。它可以从外部客户端隐藏系统的复杂性。

# 第 8 章　使用 API 网关进行路由和过滤

本章将讨论基于微服务架构的下一个重要元素，即 API 网关（API Gateway）。这不是我们在实践中第一次遇到这个元素。本书在第 4 章“服务发现”中实现了一个简单的网关模式，目的是介绍分区机制如何在使用 Eureka 的情况下进行服务发现。我们还曾经使用过 Netflix 的 Zuul 库，它是一个基于 Java 虚拟机（Java Virtual Machine，JVM）的路由器和服务器端负载均衡器。Netflix 设计了 Zuul，以提供身份验证、压力测试和金丝雀测试（Canary Test）、动态路由，以及主动流量管理/主动多区域流量管理（Active Multiregional Traffic Management）等功能。虽然没有明确说明，但它也可以充当微服务架构中的网关，其主要任务是从外部客户端隐藏系统的复杂性。

事实上，到目前为止，Zuul 在 Spring Cloud 框架内的 API 网关模式实现方面没有任何竞争对手。但是，随着一个名为 Spring Cloud Gateway 的新项目的逐步发展，情况正在发生变化。这个新项目建立在 Spring Framework 5、Project Reactor 和 Spring Boot 2.0 的基础之上。该库的最近一个稳定版本是 1.0.0，但目前正在开发的版本 2.0.0 中则有许多重要的变化，它仍然处于里程碑阶段。Spring Cloud Gateway 旨在提供一种简单而有效的方法来路由 API，并提供与其相关的跨领域问题，如安全性、监控/指标和弹性。虽然该解决方案相对较新，但绝对值得关注。

本章将要讨论的主题包括：

- 基于 URL 的静态路由和负载均衡。
- 将 Zuul 和 Spring Cloud Gateway 与服务发现集成。
- 使用 Zuul 创建自定义过滤器。
- 使用 Zuul 自定义路由配置。
- 在路由失败的情况下提供 Hystrix 后备。
- 对 Spring Cloud Gateway 中包含的主要组件——谓词和网关过滤器的说明。

## 8.1　使用 Spring Cloud Netflix Zuul

Spring Cloud 实现了一个嵌入式 Zuul 代理，允许前端应用程序对后端服务的代理调用。此功能对外部客户端非常有用，因为它隐藏了系统复杂性，有助于避免为所有微服

务独立管理跨源资源共享（Cross-Origin Resource Sharing，CORS）和身份验证问题。要启用它，可以使用@EnableZuulProxy 注解 Spring Boot 的 main 类，并将传入的请求转发到目标服务。当然，Zuul 也可以与 Ribbon 负载均衡器、Hystrix 断路器和服务发现（如使用 Eureka）集成在一起。

## 8.1.1 构建网关应用程序

现在不妨先回到第 7 章中的示例，将最后一个元素附加到基于微服务的架构：API 网关中。我们尚未考虑的是外部客户端调用服务的方式。首先，我们不希望公开系统内运行的所有微服务的网络地址。我们还可以在一个地方执行某些操作，如请求身份验证或设置跟踪标头。对于上述问题的解决方案是仅共享单个边缘网络地址，该地址会将所有传入请求代理到适当的服务。当前示例的系统架构如图 8.1 所示。

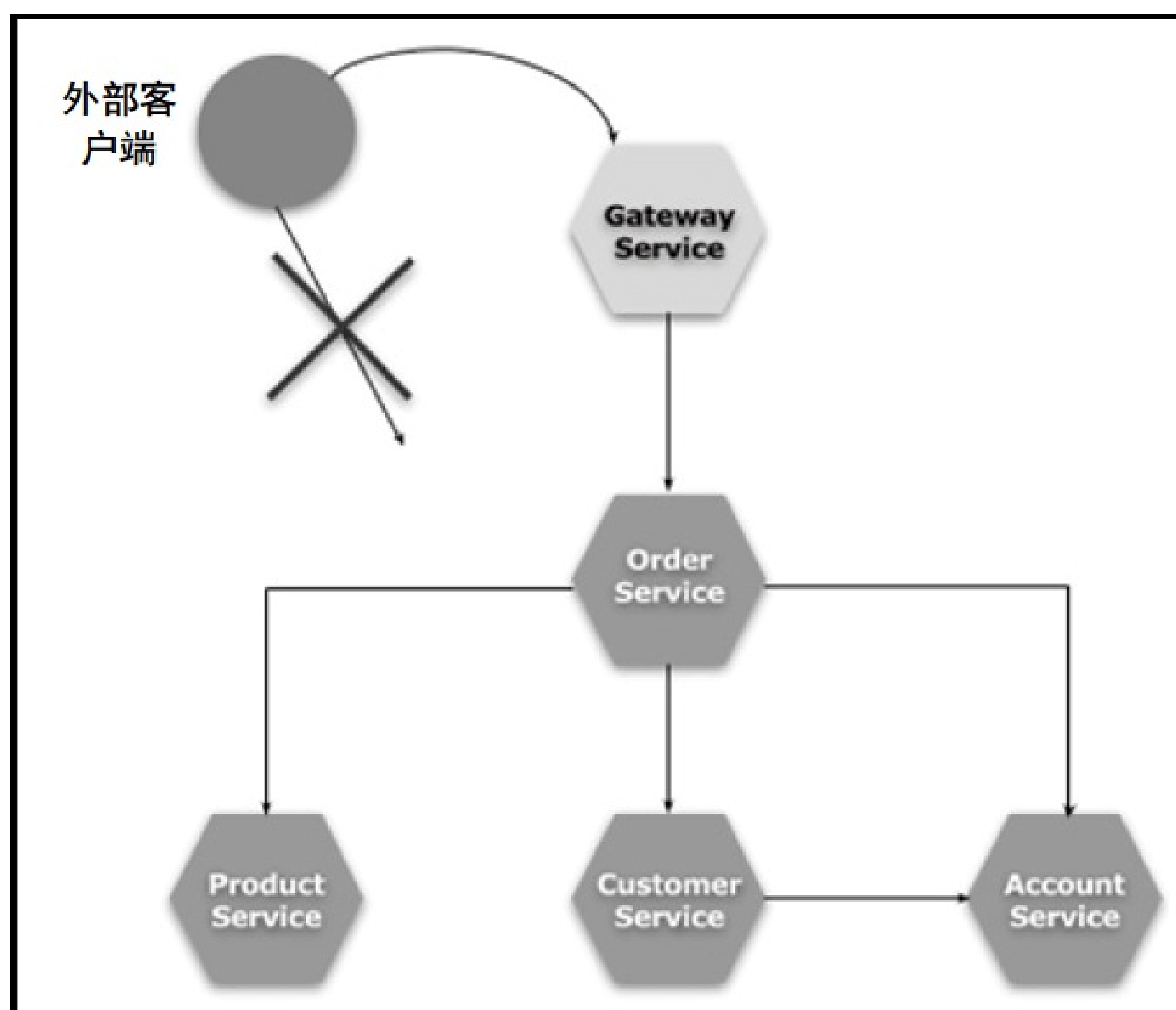



图 8.1　示例系统架构

对于当前示例的这些需求，不妨回过头来参考一下第 7 章已经讨论过的项目。它可以在 GitHub 的 master 分支（https://github.com/piomin/sample-spring-cloud-comm.git）中找到。现在，我们将为该项目添加一个名为 gateway-service 的新模块。第一步就是将 Zuul 包含在 Maven 依赖项中，这里必须使用 spring-cloud-starter-zuul 启动器。

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-zuul</artifactId>
</dependency>
```

在使用@EnableZuulProxy 注解 Spring Boot 主类之后，可以继续进行路由配置，该配置在 application.yml 文件中提供。默认情况下，Zuul 启动工件不包括服务发现客户端。路由是静态配置的，url 属性将被设置为服务的网络地址。现在，如果启动所有微服务和网关应用程序，则可以尝试通过网关调用它们。每个服务在每个路由的配置属性 path 中设置的路径下可用，如 http://localhost:8080/account/1 地址将被转发到 http://localhost:8091/1。

```
server:
 port: ${PORT:8080}

zuul:
 routes:
  account:
   path: /account/**
   url: http://localhost:8091
  customer:
   path: /customer/**
   url: http://localhost:8092
  order:
   path: /order/**
   url: http://localhost:8090
  product:
   path: /product/**
   url: http://localhost:8093
```

### 8.1.2　与服务发现集成

上一个示例中提供的静态路由配置对于基于微服务的系统来说是不够的。API 网关的主要需求是与服务发现的内置集成。要为 Zuul 启用在使用 Eureka 情况下的服务发现，开发人员必须在项目依赖项中包含 spring-cloud-starter-eureka 启动器，并通过 @EnableDiscoveryClient 来注解应用程序的 main 类，然后再启用客户端。实际上，让网关在发现服务器中注册它自己是没有意义的，它必须仅获取注册服务的当前列表。因此，开发人员可以将 eureka.client.registerWithEureka 属性设置为 false，以这种方式来禁用该注册。至于 application.yml 文件中的路由定义则非常简单。每个路由的名称都将映射到 Eureka 中的应用程序服务名称。

```
zuul:
 routes:
  account-service:
   path: /account/**
  customer-service:
   path: /customer/**
  order-service:
   path: /order/**
  product-service:
   path: /product/**
```

### 8.1.3　自定义路由配置

有若干个配置设置均允许开发人员自定义 Zuul 代理的行为。其中一些与服务发现集成有非常密切的关联。

#### 1．忽略已注册的服务

默认情况下，Spring Cloud Zuul 会公开在 Eureka 服务器中注册的所有服务。如果要跳过自动添加每项服务，则必须使用与发现服务器中所有被忽略的服务名称相匹配的模式设置 zuul.ignored-services 属性。那么它在实践中是如何运作的呢？即使开发人员没有使用 zuul.routes.*属性提供任何配置，Zuul 也会从 Eureka 获取服务列表并自动将它们绑定到具有服务名称的路径上。例如，account-service 服务将在网关地址 http://localhost:8080/account-service/**下可用。现在，如果在 application.yml 文件中设置以下配置，那么它将忽略 account-service 服务并以 HTTP 404 状态响应。

```
zuul:
  ignoredServices: 'account-service'
```

也可以通过将 zuul.ignored-services 设置为'*'来忽略所有已注册的服务。如果某服务匹配了被忽略的模式，但它也包含在路由映射配置中，那么它将包含在 Zuul 中。例如，在以下情况下，将仅处理 customer-service 服务。

```
zuul:
 ignoredServices: '*'
  routes:
   customer-service: /customer/**
```

#### 2．明确设置服务名称

开发人员还可以使用 serviceId 属性在配置中设置发现服务器的服务名称。这种方式

可以提供对路径的细粒度控制，因为这样意味着可以单独指定路径和 serviceId。以下是路由的等效配置。

```
zuul:
 routes:
  accounts:
   path: /account/**
   serviceId: account-service
  customers:
   path: /customer/**
   serviceId: customer-service
  orders:
   path: /order/**
   serviceId: order-service
  products:
   path: /product/**
   serviceId: product-service
```

### 3．使用 Ribbon 客户端进行路由定义

还有另一种配置路由的方法。开发人员可能会禁用 Eureka 发现，以便仅依赖 Ribbon 客户端的 listOfServers 属性提供的网络地址列表。默认情况下，可以通过 Ribbon 客户端在所有服务实例之间对网关的所有传入请求进行负载均衡。即使启用或禁用服务发现，此规则也适用，示例代码如下。

```
zuul:
 routes:
  accounts:
   path: /account/**
   serviceId: account-service

ribbon:
 eureka:
  enabled: false

account-service:
 ribbon:
  listOfServers: http://localhost:8091,http://localhost:9091
```

### 4．在路径中添加前缀

有时需要为通过网关调用的服务设置不同的路径，而不是允许它们直接可用。在这

种情况下，Zuul 提供了为所有定义的映射添加前缀的功能。这可以使用 zuul.prefix 属性轻松配置。默认情况下，Zuul 会在将请求转发给服务之前去掉该前缀。

但是，也可以通过将 zuul.stripPrefix 属性设置为 false 来禁用该行为。stripPrefix 属性不仅可以为所有已定义的路由全局配置，还可以为每个路由配置。

以下就是一个为所有转发的请求添加 /api 前缀的示例。举例来说，现在开发人员如果想调用 account-service 服务的 GET /{id}端点，则应该使用的地址是 http://localhost:8080/api/account/1。

```
zuul:
 prefix: /api
 routes:
  accounts:
   path: /account/**
   serviceId: account-service
  customers:
   path: /customer/**
   serviceId: customer-service
```

如果此时提供了将 stripPrefix 属性设置为 false 的配置，会出现什么问题呢？在这种情况下，Zuul 会尝试在上下文路径/api/account 和/api/customer 下查找目标服务中的端点。

```
zuul:
 prefix: /api
 stripPrefix: false
```

### 5. 连接设置和超时

Spring Cloud Netflix Zuul 的主要任务是将传入请求路由到下游服务。因此，它必须使用HTTP 客户端实现来与这些服务进行通信。Zuul 使用的默认 HTTP 客户端现在由 Apache HTTP Client 支持，而不是已经不推荐使用的 Ribbon RestClient。如果要使用 Ribbon，则应该将 ribbon.restclient.enabled 的属性设置为 true。或者也可以通过将 ribbon.okhttp.enabled 属性设置为 true 来尝试 OkHttpClient。

开发人员可以配置 HTTP 客户端的基本设置，如连接或读取超时，以及最大连接数。根据是否使用了服务发现，此类配置有两个可用选项。如果已通过 url 属性定义了具有指定网络地址的 Zuul 路由，则应设置 zuul.host.connect-timeout-millis 和 zuul.host.socket-timeout-millis。为了控制最大连接数，开发人员应该覆盖 zuul.host.maxTotalConnections 属性的默认值，该属性的默认设置为 200。还可以通过设置 zuul.host.maxPerRouteConnections 属性定义每个路由的最大连接数，该属性的默认值为 20。

如果 Zuul 已经被配置为从发现服务器获取服务列表，则需要使用 Ribbon 客户端属性 ribbon.ReadTimeout 和 ribbon.SocketTimeout 配置与以前相同的超时值。还可以使用 ribbon.MaxTotalConnections 和 ribbon.MaxConnectionsPerHost 属性自定义最大连接数。

#### 6. 保护标头的安全

如果在请求中已经设置了诸如 Authorization 之类的 HTTP 标头，但是它未被转发到下游服务，你是否会感到有些疑惑？其实这是因为 Zuul 定义了敏感标头的默认列表，这些标头在路由过程中将被删除。这个默认列表中的标头包括 Cookie、Set-Cookie 和 Authorization。此功能是站在与外部服务器进行通信的角度进行设计的。虽然不反对在同一系统中的服务之间共享标头，但出于安全原因，不建议与外部服务器共享标头。如果有必要，可以通过覆盖 sensitiveHeaders 属性的默认值来自定义此方法。它可以为所有路由全局设置，也可以仅针对单个路由设置。sensitiveHeaders 不是一个空的黑名单，所以，要让 Zuul 转发所有标头，则应该明确地将它设置为空列表。

```
zuul:
 routes:
  accounts:
   path: /account/**
   sensitiveHeaders:
   serviceId: account-service
```

### 8.1.4　管理端点

Spring Cloud Netflix Zuul 公开了两个额外的管理端点用于监控。

- Routes（路由）：打印已经定义的路由的列表。
- Filters（过滤器）：打印已经实现的过滤器的列表（可以从 Spring Cloud Netflix 的 1.4.0 版本获得）。

要启用管理端点功能，必须在项目依赖项中包含 spring-boot-starter-actuator，这和前文介绍的方法是一样的。另外，考虑到测试需要，最好能禁用端点安全性设置，方法是将 management.security.enabled 属性设置为 false。现在可以调用 GET /routes 方法，它将为我们的示例系统打印以下 JSON 响应。

```
{
  "/api/account/**": "account-service",
  "/api/customer/**": "customer-service",
  "/api/order/**": "order-service",
```

```
  "/api/product/**": "product-service",
}
```

要获得更多详细信息，必须将?format=details 查询字符串添加到/routes 路径。该选项在 Spring Cloud 1.4.0 版（Edgware 版本列车）中也可以使用。还有一种 POST /route 方法会强制刷新当前存在的路由。此外，也可以通过将 endpoints.routes.enabled 设置为 false 来禁用整个端点。

```
"/api/account/**": {
  "id": "account-service",
  "fullPath": "/api/account/**",
  "location": "account-service",
  "path": "/**",
  "prefix": "/api/account",
  "retryable": false,
  "customSensitiveHeaders": false,
  "prefixStripped": true
}
```

/filters 端点的响应结果非常有趣。开发人员可以在 Zuul 网关上查看默认情况下可用的过滤器数量和类型。以下就是使用一个选定过滤器的响应片段。它包含完整的类名、调用顺序和状态。有关过滤器的更多信息，请参阅本章第 8.1.6 节“Zuul 过滤器”。

```
"route": [{
 "class":
"org.springframework.cloud.netflix.zuul.filters.route.RibbonRoutingFilter",
 "order": 10,
 "disabled": false,
 "static": true
}, {
...
]
```

### 8.1.5　提供 Hystrix 回退 bean

开发人员可能需要为 Zuul 配置中定义的每个路由提供在电路断开情形下的回退响应。要完成此任务，应该创建一个 ZuulFallbackProvider（目前已弃用）类型的 bean 或 FallbackProvider。在该实现中，必须指定路由 ID 模式以匹配应该由回退 bean 处理的所有路由。第二步则是返回 ClientHttpResponse 接口的实现作为 fallbackResponse 方法中的响应。

以下就是一个简单的回退 bean 示例，它将每个异常映射到 HTTP 状态 200 OK，并在 JSON 响应中设置 errorCode 和 errorMessage。该回退将仅对 account-service 服务的路由执行。

```
public class AccountFallbackProvider implements FallbackProvider {

    @Override
    public String getRoute() {
        return "account-service";
    }

    @Override
    public ClientHttpResponse fallbackResponse(Throwable cause) {
        return new ClientHttpResponse() {

            @Override
            public HttpHeaders getHeaders() {
                HttpHeaders headers = new HttpHeaders();
                headers.setContentType(MediaType.APPLICATION_JSON);
                return headers;
            }

            @Override
            public InputStream getBody() throws IOException {
                AccountFallbackResponse response = new
AccountFallbackResponse("1.2", cause.getMessage());
                return new ByteArrayInputStream(new
ObjectMapper().writeValueAsBytes(response));
            }

            @Override
            public String getStatusText() throws IOException {
                return "OK";
            }

            @Override
            public HttpStatus getStatusCode() throws IOException {
                return HttpStatus.OK;
            }

            @Override
            public int getRawStatusCode() throws IOException {
```

```
                return 200;
            }

            @Override
            public void close() {

            }
        };
    }
    // ...
}
```

## 8.1.6　Zuul 过滤器

如前文所述，Spring Cloud Zuul 默认提供了若干个 bean，它们是 ZuulFilter 接口的实现。通过将 zuul.<SimpleClassName>.<filterType>.disable 属性设置为 true，可以禁用每个内置过滤器。例如，要禁用 org.springframework.cloud.netflix.zuul.filters.post.SendResponseFilter，就必须设置 zuul.SendResponse Filter.post.disable=true。

开发人员对 HTTP 过滤机制可能会很熟悉。过滤器将动态拦截请求和响应，以转换或仅使用从 HTTP 消息中获取的信息。它可以在传入请求或传出响应之前或之后触发。我们可以确定 Zuul 为 Spring Cloud 提供的以下几种类型的过滤器。

- 预处理过滤器（Pre Filter）：用于在 RequestContext 中准备初始数据，以便在下游过滤器中使用。主要职责是设置路由过滤器所需的信息。
- 路由过滤器（Route Filter）：在预过滤器之后调用，负责创建对其他服务的请求。使用它的主要原因是需要使请求或响应适应客户端所需的模型。
- 后处理过滤器（Post Filter）：最为常见，它将操纵响应，甚至可能转换响应的正文。
- 错误过滤器（Error Filter）：仅在其他过滤器抛出异常时执行。它只有一个内置的错误过滤器实现。如果 RequestContext.getThrowable()不为 null，则执行 SendErrorFilter。

### 1. 预定义的过滤器

如果使用@EnableZuulProxy 注解 main 类，则 Spring Cloud Zuul 会加载 SimpleRouteLocator 和 DiscoveryClientRouteLocator 使用的过滤器 bean。以下是作为普通 Spring bean 安装的最重要实现的列表。

- ServletDetectionFilter：这是一个预处理过滤器。它将检查请求是否通过 Spring Dispatcher，然后它会使用 FilterConstants.IS_DISPATCHER_SERVLET_REQUEST_KEY 键设置一个布尔值。
- FormBodyWrapperFilter：这是一个预处理过滤器。它将解析表单数据并为下游请求重新编码。
- PreDecorationFilter：这是一个预处理过滤器。它将根据提供的 RouteLocator 确定路由的位置和方式。它还负责设置与代理相关的标头。
- SendForwardFilter：这是一个路由过滤器。它将使用 RequestDispatcher 转发请求。
- RibbonRoutingFilter：这是一个路由过滤器。它将使用 Ribbon、Hystrix 和外部 HTTP 客户端（如 Apache HttpClient、OkHttpClient 或 Ribbon HTTP 客户端）发送请求。服务 ID 取自请求上下文。
- SimpleHostRoutingFilter：这是一个路由过滤器。它通过 Apache HTTP 客户端发送请求到 URL。URL 位于请求上下文中。
- SendResponseFilter：这是一个后处理过滤器。它会将代理请求的响应写入当前响应。

### 2. 自定义过滤器实现

除了默认安装的过滤器之外，开发人员还可以创建自定义实现。每个自定义的过滤器都必须实现 ZuulFilter 接口及其 4 种方法。这些方法负责设置过滤器的类型（filterType）、确定具有相同类型的其他过滤之间的过滤器执行顺序（filterOrder）、启用或禁用过滤器（shouldFilter）以及过滤器逻辑实现（run）。以下就是一个将 X-Response-ID 标头添加到响应的示例实现。

```
public class AddResponseIDHeaderFilter extends ZuulFilter {

    private int id = 1;

    @Override
    public String filterType() {
        return "post";
    }

    @Override
    public int filterOrder() {
        return 10;
    }
```

```
    @Override
    public boolean shouldFilter() {
        return true;
    }

    @Override
    public Object run() {
        RequestContext context = RequestContext.getCurrentContext();
        HttpServletResponse servletResponse = context.getResponse();
        servletResponse.addHeader("X-Response-ID",
         String.valueOf(id++));
        return null;
    }

}
```

这并不是全部。事实上，自定义过滤器实现也应该在 main 类或 Spring 配置类中声明为@Bean。

```
@Bean
AddResponseIDHeaderFilter filter() {
    return new AddResponseIDHeaderFilter();
}
```

## 8.2　使用 Spring Cloud Gateway

Spring Cloud Gateway 有以下 3 个基本概念。

- 路由（Route）：这是网关的基本构建块。它由用于标识路由的唯一 ID、目标 URI、谓词列表和过滤器列表组成。仅当已满足所有谓词时才匹配路径。
- 谓词（Predicate）：这些是在处理每个请求之前执行的逻辑。它负责检测 HTTP 请求的不同属性（如标头和参数）是否与定义的标准匹配。该实现基于 Java 8 接口 java.util.function.Predicate <T>。其输入类型则依次基于 Spring 的 org.springframework.web.server.ServerWebExchange。
- 过滤器（Filter）：它们允许修改传入的 HTTP 请求或传出的 HTTP 响应。可以在发送下游请求之前或之后修改它们。路径过滤器的范围限定为特定路径。它们实现了 Spring 的 org.springframework.web.server.GatewayFilter。

### 8.2.1　为项目启用 Spring Cloud Gateway

Spring Cloud Gateway 构建于 Netty Web 容器和 Reactor 框架之上。Reactor 项目和 Spring Web Flux 可以与 Spring Boot 2.0 版一起使用。到目前为止，我们使用的都是 1.5 版，因此父项目版本的声明不同。目前，Spring Boot 2.0 仍处于里程碑阶段。以下是 Maven 的 pom.xml 中的片段，它继承自 spring-boot-starter-parent 项目。

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.0.0.M7</version>
</parent>
```

与前面的示例相比，我们还需要更改 Spring Cloud 的版本列车。最新的里程碑版本是 Finchley.M5。

```
<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
<project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
    <java.version>1.8</java.version>
    <spring-cloud.version>Finchley.M5</spring-cloud.version>
</properties>
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>${spring-cloud.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

在设置了正确版本的 Spring Boot 和 Spring Cloud 之后，最终才可能在项目依赖项中包含 spring-cloud-starter-gateway 启动器。

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-gateway</artifactId>
</dependency>
```

## 8.2.2　内置谓词和过滤器

Spring Cloud Gateway 包括许多内置路由谓词（Route Predicate）和网关过滤器工厂（Gateway Filter Factory）。可以使用 application.yml 文件中的配置属性或使用 Fluent Java Routes API 以编程方式定义每个路由。表 8.1 就是其可用谓词工厂列表。可以将多个工厂组合为具有逻辑与（and）关系的单个路由定义。在 application.yml 文件中，通过 spring.cloud.gateway.routes 属性下已定义的每个路由的 predicates 属性可以配置过滤器集合。

**表 8.1　可用谓词工厂列表**

| 名　称 | 说　明 | 示　例 |
|---|---|---|
| After Route | 它需要一个日期时间参数并匹配在它之后发生的请求 | After=2017-11-20T... |
| Before Route | 它需要一个日期时间参数并匹配在它之前发生的请求 | Before=2017-11-20T... |
| Between Route | 它需要两个日期时间参数，并匹配在这些日期之间发生的请求 | Between=2017-11-20T..., 2017-11-21T... |
| Cookie Route | 它采用 cookie 名称和正则表达式参数，在 HTTP 请求的标头中查找 cookie，并将其值与提供的表达式匹配 | Cookie=SessionID, abc. |
| Header Route | 它采用标头名称和正则表达式参数，在 HTTP 请求的标头中查找特定标头，并将其值与提供的表达式匹配 | Header=X-Request-Id, \d+ |
| Host Route | 它采用一个具有.分隔符主机名 ANT 样式的模式作为参数，并将其与 Host 标头匹配 | Host=**.example.org |
| Method Route | 它需要 HTTP 方法作为参数进行匹配 | Method=GET |
| Path Route | 它采用请求上下文路径的模式作为参数 | Path=/account/{id} |
| Query Route | 它需要两个参数。一个必需的参数和一个可选的正则表达式，并将它们与查询参数相匹配 | Query=accountId, 1. |
| RemoteAddr Route | 它采用 CIDR 表示法中的 IP 地址列表，如 192.168.0.1/16，并将其与请求的远程地址进行匹配 | RemoteAddr=192.168.0.1/16 |

网关过滤器模式也有一些内置实现。表 8.2 提供了其可用工厂列表。在 application.yml 文件中，通过 spring.cloud.gateway.routes 属性下定义的每个路由的 filters 属性可以配置过滤器集合。

**表 8.2　可用过滤器工厂列表**

| 名　　称 | 说　　明 | 示　　例 |
|---|---|---|
| AddRequestHeader | 在 HTTP 请求中添加标头，并且使用在参数中提供的名称和值 | AddRequestHeader=X-Response-ID, 123 |
| AddRequestParameter | 在 HTTP 请求中添加查询参数，并且使用在参数中提供的名称和值 | AddRequestParameter=id, 123 |
| AddResponseHeader | 在 HTTP 响应中添加标头，并且使用在参数中提供的名称和值 | AddResponseHeader=X-Response-ID, 123 |
| Hystrix | 它采用一个参数，这个参数就是 HystrixCommand 的名称 | Hystrix=account-service |
| PrefixPath | 为参数中定义的 HTTP 请求路径添加前缀 | PrefixPath=/api |
| RequestRateLimiter | 它将根据 3 个输入参数限制每个用户的处理请求数。这 3 个参数包括：每秒最大请求数、突发容量和返回用户密钥的 bean | RequestRateLimiter=10,20, #{@userKeyResolver} |
| RedirectTo | 它将 HTTP 状态和重定向 URL 作为参数，并将其放入 Location HTTP 标头以执行重定向 | RedirectTo=302, http://localhost:8092 |
| RemoveNonProxyHeaders | 它将从转发的请求中删除一些逐跳（hop-by-hop）标头，如 Keep-Alive、Proxy-Authenticate 或 Proxy-Authorization | - |
| RemoveRequestHeader | 它采用标头的名称作为参数，并将其从 HTTP 请求中删除 | RemoveRequestHeader=X-Request-Foo |
| RemoveResponseHeader | 它采用标头的名称作为参数，并将其从 HTTP 响应中删除 | RemoveResponseHeader=X-Response-ID |
| RewritePath | 它采用路径 regexp 参数和替换参数，然后将重写请求的路径 | RewritePath=/account/(?<path>.*), /$\{path} |
| SecureHeaders | 它可以给响应添加一些安全标头 | - |
| SetPath | 它采用带有路径模板参数的单一参数，并且将更改请求路径 | SetPath=/{segment} |
| SetResponseHeader | 它将采用名称和值参数来设置 HTTP 响应的标头 | SetResponseHeader=X-Response-ID, 123 |
| SetStatus | 它将采用一个状态参数，该参数必须是有效的 HTTP 状态，然后将其设置在响应上 | SetStatus=401 |

以下是一个简单的示例，其中包含两个谓词和两个过滤器集合。每次 GET /account / {id}请求进入网关时都将转发到 http://localhost:8080/api/account/ {id}，其中包含新的 HTTP 标头 X-Request-ID。

```
spring:
  cloud:
    gateway:
      routes:
      - id: example route
        uri: http://localhost:8080
        predicates:
        - Method=GET
        - Path=/account/{id}
        filters:
        - AddRequestHeader=X-Request-ID, 123
        - PrefixPath=/api
```

可以使用 Route 类中定义的 Fluent API 提供相同的配置。这种风格给了开发人员更大的灵活性。虽然通过 YAML 进行配置可以使用逻辑与（and）组合谓词，但是 Fluent Java API 允许开发人员在 Predicate 类上使用 and()、or()和 negate()运算符。以下是使用 Fluent API 实现的替代路由。

```
@Bean
public RouteLocator customRouteLocator(RouteLocatorBuilder routeBuilder)
{
    return routeBuilder.routes()
        .route(r -> r.method(HttpMethod.GET).and().path("/account/{id}")
            .addRequestHeader("X-Request-ID", "123").prefixPath("/api")
            .uri("http://localhost:8080"))
        .build();
}
```

### 8.2.3　微服务的网关

现在不妨回到之前的基于微服务的系统的示例，这是本章在基于 Spring Cloud Netflix Zuul 的 API 网关配置内容中讨论过的例子。我们希望为应用程序准备静态路由定义，并且该定义与基于 Zuul 代理准备的相同。然后，每个服务将在网关地址和特定路径下可用，如 http://localhost:8080/account/**。使用 Spring Cloud Gateway 声明此类配置的最合适方式是通过 Path Route Predicate Factory 和 RewritePath GatewayFilter Factory。重写路径机制

将通过参与或添加一些模式来更改请求路径。在我们的示例中，每个传入的请求路径都将从 account/123 重写为 /123。以下是该网关的 application.yml 文件。

```
server:
 port: ${PORT:8080}

spring:
 application:
  name: gateway-service
cloud:
 gateway:
  routes:
  - id: account-service
    uri: http://localhost:8091
    predicates:
    - Path=/account/**
    filters:
    - RewritePath=/account/(?<path>.*), /$\{path}
  - id: customer-service
    uri: http://localhost:8092
    predicates:
    - Path=/customer/**
    filters:
    - RewritePath=/customer/(?<path>.*), /$\{path}
  - id: order-service
    uri: http://localhost:8090
    predicates:
    - Path=/order/**
    filters:
    - RewritePath=/order/(?<path>.*), /$\{path}
  - id: product-service
    uri: http://localhost:8093
    predicates:
    - Path=/product/**
    filters:
    - RewritePath=/product/(?<path>.*), /$\{path}
```

令人惊讶的是，这就是本示例必须要做的一切。与我们在使用其他 Spring Cloud 组件（如 Eureka 或 Config Server）时所做的相比，在这里不必提供任何其他注解。因此，我们的网关应用程序的 main 类将如以下代码所示。必须使用 mvn clean install 构建项目并使用 java -jar 启动它，或者也可以从集成开发环境运行 main 类。本示例应用程序的源代码可在 GitHub（https://github.com/piomin/sample-spring-cloud-gateway.git）上获得。

```
@SpringBootApplication
public class GatewayApplication {

    public static void main(String[] args) {
        SpringApplication.run(GatewayApplication.class, args);
    }

}
```

## 8.2.4　与服务发现集成

网关可以被配置为基于在服务发现中注册的服务列表来创建路由。它可以与具有 DiscoveryClient 兼容服务注册表的解决方案集成，如 Netflix Eureka、Consul 或 Zookeeper。要启用 DiscoveryClient 路由定义定位器，应将 spring.cloud.gateway.discovery.locator.enabled 属性设置为 true，并在类路径上提供 DiscoveryClient 实现。开发人员可以使用 Eureka 客户端和服务器进行发现。请注意，要使用 Spring Cloud 的最新里程碑版本 Finchley.M5，在该版本中，所有 Netflix 的工件名称都已更改，例如，现在 spring-cloud-starter-eureka 已经被修改为 spring-cloud-starter- netflix-eureka-client。

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
```

Eureka 客户端应用程序的 main 类应该相同，使用@DiscoveryClient 注解。以下是带路由配置的 application.yml 文件。与前一个示例相比，唯一的变化是每个已定义路由的 uri 属性。开发人员可以使用从具有 lb 前缀的发现服务器（如 lb://order-service）获取的名称，而不是提供网络地址。

```
spring:
 application:
  name: gateway-service
 cloud:
  gateway:
   discovery:
    locator:
     enabled: true
   routes:
   -id: account-service
     uri: lb://account-service
```

```
    predicates:
    - Path=/account/**
    filters:
    - RewritePath=/account/(?<path>.*), /$\{path}
  - id: customer-service
    uri: lb://customer-service
    predicates:
    - Path=/customer/**
    filters:
    - RewritePath=/customer/(?<path>.*), /$\{path}
  - id: order-service
    uri: lb://order-service
    predicates:
    - Path=/order/**
    filters:
    - RewritePath=/order/(?<path>.*), /$\{path}
  - id: product-service
    uri: lb://product-service
    predicates:
    - Path=/product/**
    filters:
    - RewritePath=/product/(?<path>.*), /$\{path}
```

## 8.3　小　　结

在介绍完 API 网关之后，本书就已经完成了关于 Spring Cloud 中基于微服务的架构的核心元素实现的讨论。阅读完本书的这一部分之后，开发人员应该能够自定义和使用 Eureka、Spring Cloud Config、Ribbon、Feign 和 Hystrix 等工具，最后还可以使用基于 Zuul 和 Spring Cloud Gateway 的网关。

本章可以被视为两种可用解决方案的比较——较旧的 Netflix Zuul 和最新的解决方案 Spring Cloud Gateway。其中，第二种解决方案仍在不断变化。它的当前版本 2.0 只能在 Spring 5 中使用，并且在发行版中仍然不可用。第一种解决方案 Netflix Zuul 则是稳定的，但它不支持异步和非阻塞连接等特性。它仍然基于 Netflix Zuul 1.0，当然也有一个新版本的 Zuul 支持异步通信。不管它们之间的区别如何，本章已经描述了如何使用这两种解决方案提供简单和更高级的配置。基于前面章节中的示例，本章还介绍了与服务发现、客户端负载均衡器和断路器的集成。

# 第 9 章　分布式日志记录和跟踪

在将一体化应用程序分解为微服务时，开发人员通常会花费大量的时间考虑业务边界或应用程序逻辑的分区，却忘记了日志。根据笔者自己作为开发人员和软件架构师的经验，一方面，可以说很多开发人员通常都不太关注日志记录；而另一方面，负责应用程序维护的操作团队却主要依赖于日志。无论开发人员所关注的领域如何，也无论他们基于的是一体化应用程序还是微服务架构，所有应用程序都必须执行日志记录，这是无可争辩的。但是，微服务强制为应用程序日志的设计和排列添加了一个全新的维度，因为会有许多小型、独立、水平扩展的互相通信的服务在多台机器上运行；同时，通常会有大量的请求由多个服务处理。开发人员必须将这些请求关联在一起，并将所有日志存储在一个中心位置，以便更容易查看它们。Spring Cloud 引入了一个专用库，实现了分布式跟踪解决方案 Spring Cloud Sleuth。

这里还有一件事需要讨论。日志记录（Logging）与跟踪（Tracing）不一样，开发人员有必要了解一下它们之间的差异。跟踪是指跟踪程序的数据流。技术支持团队通常使用它来诊断问题发生的位置。对于开发人员来说，只有在出现错误时才必须跟踪系统流以发现性能“瓶颈”或时间。日志记录用于错误报告和检测。与跟踪相反，它应该始终启用。当设计一个大型系统并希望跨机器进行良好而灵活的错误报告时，一定要考虑以集中方式收集日志数据。推荐和最流行的解决方案是 ELK（Elasticsearch + Logstash + Kibana）堆栈。Spring Cloud 中没有用于此堆栈的专用库，但可以使用 Java 日志框架（如 Logback 或 Log4j）实现集成。Zipkin 将在本章中讨论另一种工具。它是一种典型的跟踪工具，可帮助收集可用于解决微服务架构中的延迟问题的时序数据。

本章将要讨论的主题包括：

- 基于微服务的系统中日志记录的最佳实践。
- 使用 Spring Cloud Sleuth 将跟踪信息附加到消息并关联到事件。
- 集成 Spring Boot 应用程序和 Logstash。
- 使用 Kibana 显示和过滤日志条目。
- 使用 Zipkin 作为分布式跟踪工具，并通过 Spring Cloud Sleuth 将其与应用程序集成。

## 9.1　微服务的最佳日志记录实践

处理日志记录最重要的最佳实践之一是跟踪所有传入请求和传出响应。也许这对于

部分开发人员来说是显而易见的，但我们也曾经看到过一些不符合该要求的应用程序。如果满足此需求，则基于微服务的架构将会产生一个后果，即与没有消息传递的单一应用程序相比，该系统中的日志总数会有所增加。这反过来也会要求开发人员比以前更加关注日志。我们应该尽可能地生成尽可能少的信息，即使这些信息可以告诉我们很多情况。如何实现这一目标？首先，在所有微服务中使用相同的日志消息格式就不失为一个良策。例如，可以考虑如何在应用程序日志中打印变量。鉴于通常在微服务之间交换的消息会使用 JSON 进行格式化，所以建议开发人员使用 JSON 表示法。此格式具有非常简单的标准，使日志易于阅读和解析。以下就是一个日志的片段。

```
17:11:53.712   INFO   Order received:
{"id":1,"customerId":5,"productId":10}
```

上面的格式显然比以下格式更容易分析。

```
17:11:53.712   INFO   Order received with id 1, customerId 5 and productId
10.
```

但是一般来说，最重要的是标准化。无论选择哪一种格式，在什么地方使用它才至关重要。开发人员还应该小心确保日志有意义，尽量避免不包含任何信息的句子。例如，从以下格式中完全看不出来正在处理哪个订单。

```
17:11:53.712   INFO   Processing order
```

但是，如果确实需要这种日志条目格式，则可以尝试将其分配给不同的日志级别。使用相同级别的 INFO 记录所有内容确实是一种不好的做法。某些类型的信息比其他信息更重要，因而这里的一个难点是确定应记录日志条目的级别。以下是一些建议。

- 跟踪（TRACE）：这是非常详细的信息，仅用于开发模式。可以在部署到生产环境之后将其保留一小段时间，并将其视为临时文件。
- 调试（DEBUG）：在此级别将记录程序中发生的任何事情。这主要用于开发人员的调试或故障排除。DEBUG 和 TRACE 之间的区别可能是最困难的。
- 信息（INFO）：在此级别应记录操作期间最重要的信息。这些消息必须易于理解，不仅适用于开发人员，也适用于管理员或高级用户，以便让他们快速了解应用程序正在执行的操作。
- 警告（WARN）：在此级别将记录可能会出错的所有事件。这样的过程可能会继续，但开发人员应该格外小心。
- 错误（ERROR）：通常会在此级别打印异常。这里重要的是不要在所有地方抛出异常，例如，如果只有一个业务逻辑执行没有成功，则不应该影响整个程序。
- 致命（FATAL）：此 Java 日志记录级别指定可能导致应用程序终止的非常严重

的错误事件。

虽然可能还有其他一些很好的日志记录实践，但我们已经提到的都是在基于微服务的系统中使用的最重要的日志实践。关于日志记录，还有一个方面值得一提，那就是规范化。如果开发人员希望轻松理解和解释自己的日志，则应该清楚地了解它们的收集方式和时间、它们包含的内容以及它们释出的原因。应该在所有微服务中规范化一些特别重要的特征，如 Time（发生的时间）、Hostname（发生的主机名）和 AppName（发生的程序名）。正如 9.2 节所示，当在系统中实现集中收集日志的方法时，这种规范化非常有用。

## 9.2　使用 Spring Boot 记录日志

Spring Boot 将使用 Apache Commons Logging 进行内部日志记录，但是，如果要包含启动器的依赖项，则默认情况下将在应用程序中使用 Logback。它不会抑制以任何方式使用其他日志框架的可能性。还为 Java Util Logging、Log4J2 和 SLF4J 提供了默认配置。可以在 application.yml 文件中使用 logging.*属性配置日志记录设置。默认日志输出包含以毫秒为单位的日期和时间、日志级别、进程 ID、线程名称、已发出条目的类的全名以及消息。可以通过分别对控制台和文件追加程序使用 logging.pattern.console 和 logging.pattern.file 属性来覆盖它。

默认情况下，Spring Boot 仅记录到控制台。除了控制台输出之外，要允许写入日志文件，则应该设置 logging.file 或 logging.path 属性。如果指定 logging.file 属性，则日志将在相对于当前目录的确切位置写入文件。如果设置 logging.path，则会在指定目录中创建 spring.log 文件。达到 10MB 后，日志文件将被轮换（Rotate）。

application.yml 设置文件中可以自定义的最后一件事是日志级别。默认情况下，Spring Boot 会使用 ERROR、WARN 和 INFO 级别写入消息。我们可以使用 logging.level.*属性为每个包或类覆盖此设置。

也可以使用 logging.level.root 配置根日志记录器（Root Logger）。以下是 application.yml 文件中的示例配置，它更改了默认模式格式以及一些日志级别，并设置了日志文件的位置。

```
logging:
 file: logs/order.log
 level:
  com.netflix: DEBUG
  org.springframework.web.filter.CommonsRequestLoggingFilter: DEBUG
 pattern:
```

```
  console: "%d{HH:mm:ss.SSS} %-5level %msg%n"
  file: "%d{HH:mm:ss.SSS} %-5level %msg%n"
```

正如上例所示，这样的配置非常简单，但在某些情况下，这还不够。如果要定义其他追加器（Appender）或过滤器，则应明确包括其中一个可用日志记录系统的配置。这样的日志系统如 Logback（logback-spring.xml）、Log4j2（log4j2-spring.xml）或 Java Util Logging（logging.properties）。如前文所述，默认情况下，Spring Boot 将使用 Logback 作为应用程序日志。如果在类路径的根目录中提供 logback-spring.xml 文件，它将覆盖 application.yml 中定义的所有设置。例如，开发人员可以创建每天轮换日志的文件追加器，并保留最多 10 天的历史记录。

此功能在应用程序中非常实用。9.3 节将会介绍到，在将微服务与 Logstash 集成时，便需要一个自定义的追加器。以下是 Logback 配置文件的示例片段，该片段为 logs/order.log 文件设置了每日滚动策略。

```
<configuration>
 <appender name="FILE"
class="ch.qos.logback.core.rolling.RollingFileAppender">
  <file>logs/order.log</file>
  <rollingPolicy
class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
   <fileNamePattern>order.%d{yyyy-MM-dd}.log</fileNamePattern>
   <maxHistory>10</maxHistory>
   <totalSizeCap>1GB</totalSizeCap>
  </rollingPolicy>
  <encoder>
   <pattern>%d{HH:mm:ss.SSS} %-5level %msg%n</pattern>
  </encoder>
 </appender>
 <root level="DEBUG">
  <appender-ref ref="FILE" />
 </root>
</configuration>
```

值得一提的是，Spring 建议为 Logback 使用 logback-spring.xml 而不是默认的 logback.xml。Spring Boot 包含一些 Logback 扩展，可能对高级配置有所帮助。它们不能在标准 logback.xml 中使用，而只能在 logback-spring.xml 中使用。我们列出了一些扩展，允许开发人员从 Spring 环境定义特定于配置文件的配置或接口属性。

```
<springProperty scope="context" name="springAppName"
source="spring.application.name" />
<property name="LOG_FILE" value="${BUILD_FOLDER:-build}/${springAppName}"/>
```

```
<springProfile name="development">
...
</springProfile>

<springProfile name="production">
 <appender name="flatfile"
class="ch.qos.logback.core.rolling.RollingFileAppender">
  <file>${LOG FILE}</file>
  <rollingPolicy
class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
   <fileNamePattern>${LOG FILE}.%d{yyyy-MM-dd}.gz</fileNamePattern>
   <maxHistory>7</maxHistory>
  </rollingPolicy>
  <encoder>
   <pattern>${CONSOLE LOG PATTERN}</pattern>
   <charset>utf8</charset>
  </encoder>
 </appender>
 ...
</springProfile>
```

## 9.3　使用 ELK Stack 集中日志

ELK 是 3 个开源工具——Elasticsearch、Logstash 和 Kibana 的首字母缩写，它也被称为弹性堆栈（Elastic Stack）。该系统的核心是 Elasticsearch，这是一个基于另一个用 Java 编写的开源项目 Apache Lucene 的搜索引擎。该库特别适用于需要跨平台环境中的全文搜索的应用程序。Elasticsearch 流行的主要原因是它的性能。当然，它还具有其他一些优点，如可伸缩性、灵活性和易于集成，因为它可以提供基于 JSON 的 RESTful API 来搜索已存储的数据。它有一个庞大的网络讨论社区和许多用例，但对我们来说最有趣的是它能够存储和搜索应用程序生成的日志。记录日志是在 ELK Stack 中包含 Logstash 的主要原因。这个开源数据处理管道允许开发人员收集、处理和输入数据到 Elasticsearch。

Logstash 支持许多从外部源提取事件的输入。有趣的是它有很多输出，而 Elasticsearch 只是其中之一。例如，它可以将事件写入 Apache Kafka、RabbitMQ 或 MongoDB，它可以将指标数据写入 InfluxDB 或 Graphite。它不仅可以接收数据并将数据转发到目的地，还可以动态解析和转换数据。

Kibana 是 ELK Stack 的最后一个元素。它是 Elasticsearch 的开源数据可视化插件。它允许开发人员可视化、探索和发现 Elasticsearch 中的数据。开发人员可以通过创建搜索查询轻松显示和过滤从应用程序收集到的所有日志。在此基础上，开发人员还可以将数据导出为 PDF 或 CSV 格式以提供报告。

### 9.3.1　在机器上设置 ELK 堆栈

在尝试将任何日志从应用程序发送到 Logstash 之前，开发人员必须在本地计算机上配置 ELK Stack。最合适的运行方式是通过 Docker 容器。堆栈中的所有产品都可用作 Docker 镜像。有一个专门的 Docker 注册表由 Elastic Stack 的供应商托管。有关已发布镜像和标签的完整列表，请访问 www.docker.elastic.co 查找。所有这些都将使用 centos:7 作为基本镜像。

我们将从 Elasticsearch 实例开始。可以使用以下命令启动它的开发。

```
docker run -d --name es -p 9200:9200 -p 9300:9300 -e
"discovery.type=single-node"
docker.elastic.co/elasticsearch/elasticsearch:6.1.1
```

在开发模式下运行 Elasticsearch 是最方便的运行方式，因为我们不必提供任何其他配置。如果要在生产模式下启动它，则需要将 vm.max_map_count Linux 内核设置为至少 262144。修改它的过程因操作系统平台而异。对于带有 Docker Toolbox 的 Windows 系统来说，必须通过 docker-machine 设置。

```
docker-machine ssh
sudo sysctl -w vm.max_map_count=262144
```

下一步是使用 Logstash 运行容器。除了使用 Logstash 启动容器之外，开发人员还应该定义输入和输出。输出很明显，就是 Elasticsearch，现在可以在默认的 Docker 机器地址 192.168.99.100 下使用它。作为输入，我们定义了简单的 TCP 插件 logstash-input-tcp，它与我们的示例应用程序中用作日志记录追加器的 LogstashTcpSocketAppender 兼容。我们的微服务中的所有日志都将以 JSON 格式发送。所以，现在为该插件设置 json 编解码器非常重要。每个微服务都将使用其名称和 micro 前缀（显然这是为了表示它是微服务）在 Elasticsearch 中编制索引。以下是 Logstash 配置文件 logstash.conf。

```
input {
  tcp {
    port => 5000
    codec => json
```

```
  }
}

output {
  elasticsearch {
    hosts => ["http://192.168.99.100:9200"]
    index => "micro-%{appName}"
  }
}
```

以下是运行 Logstash 并在端口 5000 上公开它的命令。它还会将具有上述设置的文件复制到容器并覆盖 Logstash 配置文件的默认位置。

```
docker run -d --name logstash -p 5000:5000 -v ~/logstash.conf:/config-
dir/logstash.conf docker.elastic.co/logstash/logstash-oss:6.1.1 -f
/config- dir/logstash.conf
```

最后，开发人员可以运行堆栈的最后一个元素 Kibana。默认情况下，它将在端口 5601 上公开，并连接到端口 9200 上可用的 Elasticsearch API，以便能够从那里加载数据。

```
docker run -d --name kibana -e
"ELASTICSEARCH_URL=http://192.168.99.100:9200" -p 5601:5601
docker.elastic.co/kibana/kibana:6.1.1
```

如果开发人员想在 Windows 系统的 Docker 机器上运行所有 Elastic Stack 产品，则可能需要将 Linux 虚拟映像的默认 RAM 内存增加到最小 2GB。启动所有容器之后，开发人员最终可以访问 http://192.168.99.100:5601 下的 Kibana 仪表板，然后继续将应用程序与 Logstash 集成。

## 9.3.2　将应用程序与 ELK Stack 集成

通过 Logstash 将 Java 应用程序与 ELK Stack 集成的方法有很多种。其中一种方法涉及使用 Filebeat，它是本地文件的日志数据发送器。此方法需要为 Logstash 实例配置的 beats（logstash-input-beats）输入，这实际上是默认选项。开发人员还应该在服务器计算机上安装并启动 Filebeat 守护程序。它负责将日志传递给 Logstash。

就个人而言，笔者更喜欢基于 Logback 和专用追加器的配置。它似乎比使用 Filebeat 代理更简单。除了必须部署其他服务外，Filebeat 还要求我们使用解析表达式，如 Grok 过滤器。使用 Logback 追加器时，不需要任何日志发送程序。这个追加器在项目 Logstash JSON 编码器中可用。开发人员可以通过在 logback-spring.xml 文件中声明 net.logstash.logback.appender.LogstashSocketAppender appender 来为自己的应用程序启用它。

我们还将讨论使用消息代理将数据发送到 Logstash 的替代方法。在我们即将讨论的示例中，将演示如何使用 Spring AMQPAppender 将日志事件发布到 RabbitMQ 交换消息。在这种情况下，Logstash 将订阅该消息并使用已发布的消息。

### 1．使用 LogstashTCPAppender

库 logstash-logback-encoder 可以提供 3 种类型的追加器——UDP、TCP 和异步（Async）。TCP 追加器是最常用的。值得一提的是，TCP 追加器是异步的，所有的编码和通信都被委托给一个线程。除了追加器之外，该库还提供了一些编码器和布局，使开发人员能够以 JSON 格式记录日志。因为 Spring Boot 默认包含一个 Logback 库，以及 spring-boot-starter-web，所以我们只需要为 Maven 的 pom.xml 添加一个依赖项即可。

```
<dependency>
 <groupId>net.logstash.logback</groupId>
 <artifactId>logstash-logback-encoder</artifactId>
 <version>4.11</version>
</dependency>
```

下一步是在 Logback 配置文件中使用 LogstashTCPAppender 类定义追加器。每个 TCP 追加器都要求开发人员配置编码器。可以在 LogstashEncoder 和 LoggingEventCompositeJsonEncoder 之间做出选择。

LoggingEventCompositeJsonEncoder 可以为开发人员提供更大的灵活性。它由一个或多个映射到 JSON 输出的 JSON 提供程序组成。默认情况下，如果没有配置 JSON 提供程序，那么它不会按 LogstashTCPAppender 的方式工作。它默认包含若干个标准字段，如时间戳、版本、日志程序名称和堆栈跟踪等。此外，它还会添加来自映射诊断上下文（Mapped Diagnostic Context，MDC）的所有条目和上下文，当然，开发人员也可以通过将 includeMdc 或 includeContext 属性中的一个设置为 false 来禁用它。

```
<appender name="STASH"
class="net.logstash.logback.appender.LogstashTcpSocketAppender">
 <destination>192.168.99.100:5000</destination>
 <encoder
class="net.logstash.logback.encoder.LoggingEventCompositeJsonEncoder">
  <providers>
   <mdc />
   <context />
   <logLevel />
   <loggerName />
   <pattern>
    <pattern>
```

```
    {
    "appName": "order-service"
    }
    </pattern>
   </pattern>
   <threadName />
   <message />
   <logstashMarkers />
   <stackTrace />
  </providers>
 </encoder>
</appender>
```

现在再回来看一看我们的示例系统。开发人员仍然可以在同一个 Git 存储库（https://github.com/piomin/sample-spring-cloud-comm.git）和 feign_with_discovery 分支（https://github.com/piomin/sample-spring-cloud-comm/tree/feign_with_discovery）找到该示例。笔者已根据第 9.1 节“微服务的最佳日志记录实践”中提出的建议在源代码中添加了一些日志记录条目。以下是 order-service 服务中 POST 方法的当前版本。笔者已经通过从 org.slf4j.LoggerFactory 调用 getLogger 方法将 SLF4J 上的 Logback 用作日志记录程序。

```
@PostMapping
public Order prepare(@RequestBody Order order) throws
JsonProcessingException {
    int price = 0;
    List<Product> products =
productClient.findByIds(order.getProductIds());
    LOGGER.info("Products found: {}", mapper.writeValueAsString(products));
    Customer customer =
customerClient.findByIdWithAccounts(order.getCustomerId());
    LOGGER.info("Customer found: {}", mapper.writeValueAsString(customer));
    for (Product product : products)
        price += product.getPrice();
    final int priceDiscounted = priceDiscount(price, customer);
    LOGGER.info("Discounted price: {}",
mapper.writeValueAsString(Collections.singletonMap("price",
priceDiscounted)));
    Optional<Account> account = customer.getAccounts().stream().filter(a ->
(a.getBalance() > priceDiscounted)).findFirst();
    if (account.isPresent()) {
        order.setAccountId(account.get().getId());
        order.setStatus(OrderStatus.ACCEPTED);
        order.setPrice(priceDiscounted);
```

```
            LOGGER.info("Account found: {}",
mapper.writeValueAsString(account.get()));
        } else {
            order.setStatus(OrderStatus.REJECTED);
            LOGGER.info("Account not found: {}",
mapper.writeValueAsString(customer.getAccounts()));
        }

        return repository.add(order);
}
```

现在来看一看如图 9.1 所示的 Kibana 仪表板。它可以在 http://192.168.99.100:5601 处获得。在该仪表板中可以轻松发现和分析应用程序日志。在页面左侧的菜单中可以选择所需的索引名称（在图 9.1 的屏幕截图中标记为 1）。日志统计信息显示在时间线图（标记为 2）上。可以通过单击具体条或选择一组条来缩小搜索参数所采用的时间。给定时间段内的所有日志都显示在图表下方的面板上（标记为 3）。




图 9.1　Kibana 仪表板

可以扩展每个条目以查看其详细信息。在详细的 Table（表）视图中，我们可以看到诸如 Elasticsearch 索引（_index）的名称以及微服务的级别或名称（appName）。大多数这些字段都已经通过 LoggingEventCompositeJsonEncoder 设置。我们只定义了一个特定于应用程序的字段 appName，如图 9.2 所示。

December 29th 2017, 10:27:48.410　Products found: [{"id":3,"name":"Test3","price":2000},{"id":8,"name":"Test8","price":1250}]

Table　JSON

| | | |
|---|---|---|
| t | @metdata.ip_address | 192.168.99.1 |
| ⊙ | @timestamp | December 29th 2017, 10:27:48.410 |
| t | @version | 1 |
| t | _id | kiGZoWABfCW_oGSVU5XP |
| t | _index | micro-order-service |
| # | _score | - |
| t | _type | doc |
| t | appName | order-service |
| t | host | 192.168.99.1 |
| t | level | INFO |
| t | logger_name | pl.piomin.services.order.controller.OrderController |
| t | message | Products found: [{"id":3,"name":"Test3","price":2000},{"id":8,"name":"Test8","price":1250}] |
| # | port | 52,803 |
| t | thread_name | http-nio-8090-exec-2 |

图 9.2　查看 Table（表）视图

Kibana 提供了搜索特定条目的强大功能。开发人员可以仅通过单击所选条目来定义过滤器，以便定义一组搜索条件。在图 9.2 中，可以看到如何使用传入的 HTTP 请求过滤掉所有条目。如前文所述，org.springframework.web.filter.CommonsRequestLoggingFilter 类负责记录这些日志。我们刚刚定义了一个过滤器，其名称等于完全限定的日志记录器类名称。如图 9.3 所示是笔者的 Kibana 仪表板屏幕，它显示了仅由 CommonsRequestLoggingFilter 生成的日志。



图 9.3　使用过滤器生成的日志

### 2. 使用 AMQP 追加器和消息代理

使用 Spring AMQP 追加器和消息代理的配置比使用简单 TCP 追加器的方法要稍微复杂一些。首先，开发人员需要在本地计算机上启动消息代理。本书已经在第 5 章“使用 Spring Cloud Config 进行分布式配置”中详细介绍了此过程，其中还专门介绍了 RabbitMQ，以便使用 Spring Cloud Bus 重新加载动态配置。如果开发人员已在本地或作为 Docker 容器启动了 RabbitMQ 实例，则可以继续进行配置。必须创建一个队列来发布传入事件，然后将其绑定到交换消息（Exchange）。

要实现此目的，应该先登录 Rabbit 管理控制台，然后转到 Queues（队列）部分。我们已经创建了一个名为 q_logstash 的队列，并且使用名称 ex_logstash 定义了新的 Exchange 消息，如图 9.4 所示。对于所有示例微服务，队列都已经使用路由键值绑定到该 Exchange 消息。

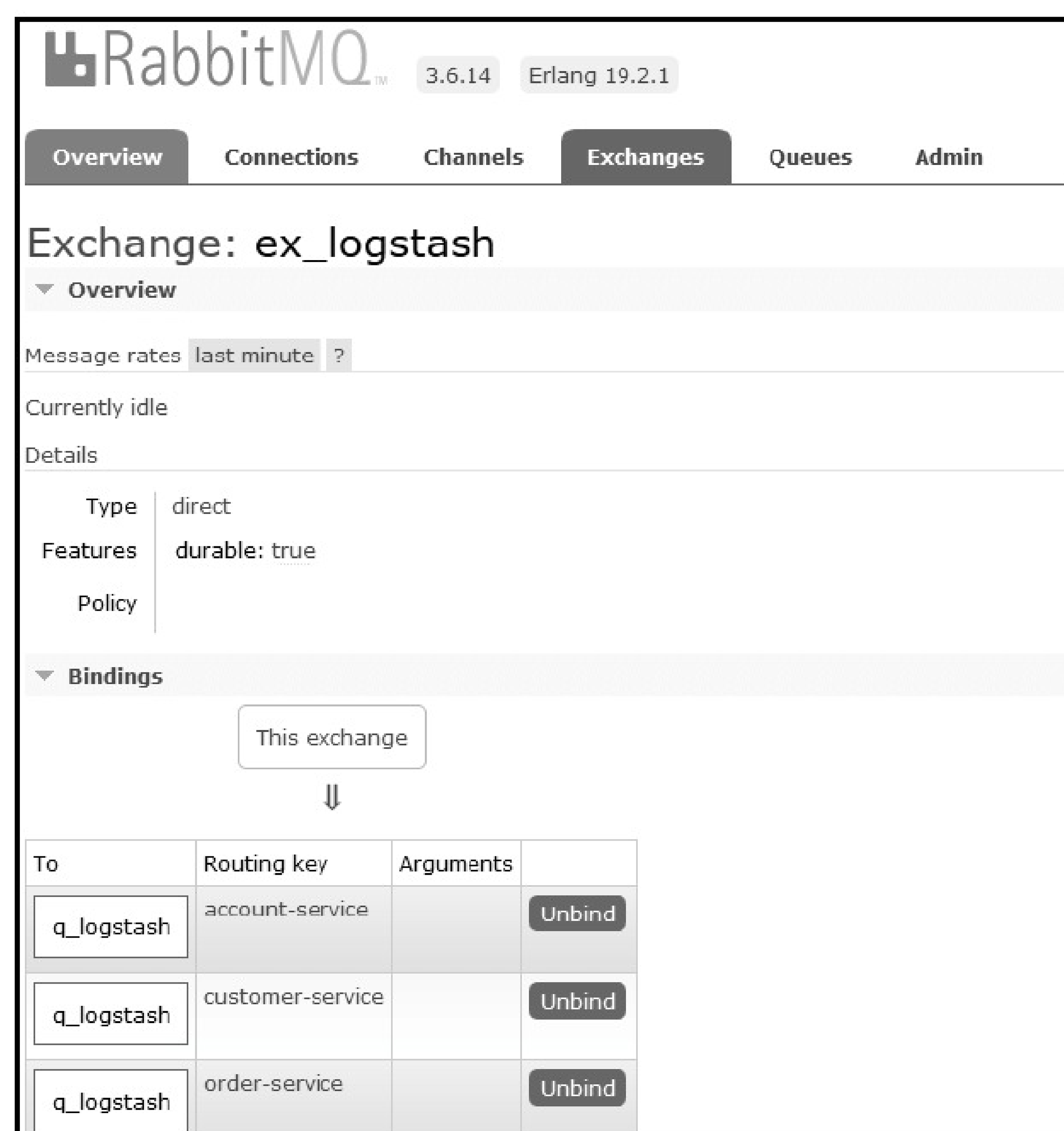



图 9.4　新定义的 Exchange 消息

在启动并配置了 RabbitMQ 的实例之后，即可开始在应用程序端集成。首先，开发人

员必须在项目依赖项中包含 spring-boot-starter-amqp，以提供 AMQP 客户端和 AMQP 追加器的实现。

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-amqp</artifactId>
</dependency>
```

然后，唯一要做的就是使用 Logback 配置文件中的 org.springframework.amqp.rabbit.logback.AmqpAppender 类来定义追加器。需要设置的最重要的属性是 RabbitMQ 网络地址（host 和 port）、已声明的交换消息名称（exchangeName）和路由键值（routingKeyPattern），该键值必须和已声明交换消息绑定的键值之一匹配。与 TCP 追加器相比，这种方法的缺点是需要自己准备发送给 Logstash 的 JSON 消息。以下是 order-service 服务的 Logback 配置的一个片段。

```
<appender name="AMQP"
 class="org.springframework.amqp.rabbit.logback.AmqpAppender">
 <layout>
  <pattern>
  {
  "time": "%date{ISO8601}",
  "thread": "%thread",
  "level": "%level",
  "class": "%logger{36}",
  "message": "%message"
  }
  </pattern>
 </layout>
 <host>192.168.99.100</host>
 <port>5672</port>
 <username>guest</username>
 <password>guest</password>
 <applicationId>order-service</applicationId>
 <routingKeyPattern>order-service</routingKeyPattern>
 <declareExchange>true</declareExchange>
 <exchangeType>direct</exchangeType>
 <exchangeName>ex_logstash</exchangeName>
 <generateId>true</generateId>
 <charset>UTF-8</charset>
 <durable>true</durable>
```

```
  <deliveryMode>PERSISTENT</deliveryMode>
</appender>
```

Logstash 可以通过声明 rabbitmq（logstash-input-rabbitmq）输入轻松地与 RabbitMQ 集成。

```
input {
  rabbitmq {
    host => "192.168.99.100"
    port => 5672
    durable => true
    exchange => "ex_logstash"
  }
}

output {
  elasticsearch {
    hosts => ["http://192.168.99.100:9200"]
  }
}
```

## 9.4　Spring Cloud Sleuth

Spring Cloud Sleuth 是一个相当小的简单项目，它为日志记录和跟踪提供了一些有用的功能。如果仔细研究“使用 LogstashTCPAppender”小节中讨论的示例，就可以很容易地看出它不可能过滤与单个请求相关的所有日志。在基于微服务的环境中，在处理进入系统的请求时，关联应用程序交换的消息也非常重要。这是创建 Spring Cloud Sleuth 项目的主要动机。

如果为应用程序启用了 Spring Cloud Sleuth，它会向请求添加一些 HTTP 标头，这允许开发人员将请求与响应和已交换的消息链接起来，这个交换是由独立应用程序完成的，而链接则可以通过 RESTful API 之类的接口完成。它定义了两个基本的工作单位：跨度（Span）和跟踪（Trace）。每一个单位都由唯一的 64 位 ID 标识。跟踪 ID 的值等于跨度 ID 的初始值。跨度是指单个交换，其中的响应将作为对请求的反应而发送。跟踪通常称为关联 IT（Correlation IT），它将帮助开发人员链接来自不同应用程序的所有日志，而这些日志是在处理进入系统的请求期间生成的。

每个跟踪和跨度 ID 都会添加到 Slf4J 映射诊断上下文（MDC）中，因此，可以在日

志聚合器（Log Aggregator）中提取具有给定跟踪或跨度的所有日志。MDC 只是一个存储当前线程上下文数据的映射。进入服务器的每个客户端请求都由不同的线程处理。由于这个原因，每个线程都可以在线程生命周期内访问其 MDC 的值。除了 spanId 和 traceId 之外，Spring Cloud Sleuth 还向 MDC 添加了以下两个跨度。

- appName：生成日志条目的应用程序的名称。
- exportable：指定是否应将日志导出到 Zipkin。

除了上述功能之外，Spring Cloud Sleuth 还可以提供以下功能。

- 对常见分布式跟踪数据模型的抽象，这允许与 Zipkin 的集成。
- 记录计时信息，以帮助进行延迟分析。它还包括不同的采样策略以管理导出到 Zipkin 的数据量。
- 与参与通信的常见 Spring 组件集成，如 servlet 过滤器、异步端点、RestTemplate、消息通道、Zuul 过滤器和 Feign 客户端等。

## 9.4.1　将 Sleuth 与应用程序集成

要为应用程序启用 Spring Cloud Sleuth 功能，只需将 spring-cloud-starter-sleuth 启动器添加到依赖项即可。

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-sleuth</artifactId>
</dependency>
```

包含此依赖项之后，应用程序生成的日志条目的格式已更改。具体如下所示。

```
2017-12-30 00:21:31.639
INFO [order-service,9a3fef0169864e80,9a3fef0169864e80,false]
49212 --- [nio-8090- exec-6]
p.p.s.order.controller.OrderController :
Products found:[{"id":2,"name":"Test2","price":1500},
{"id":9,"name":"Test9","price":2450}]
2017-12-30 00:21:31.683
INFO [order-service,9a3fef0169864e80,9a3fef0169864e80,false]
49212 --- [nio-8090- exec-6]
p.p.s.order.controller.OrderController :
Customer found:{"id":2,"name":"Adam Smith",
"type":"REGULAR","accounts":
[{"id":4,"number":"1234567893","balance":5000},
```

```
{"id":5,"number":"1234567894","balance":0},
{"id":6,"number":"1234567895","balance":5000}]}
2017-12-30 00:21:31.684
INFO [order-service,9a3fef0169864e80,9a3fef0169864e80,false]
49212 --- [nio-8090- exec-6]
p.p.s.order.controller.OrderController :
Discounted price:{"price":3752}
2017-12-30 00:21:31.684
INFO [order-service,9a3fef0169864e80,9a3fef0169864e80,false]
49212 --- [nio-8090- exec-6]
p.p.s.order.controller.OrderController :
Account found:{"id":4,"number":"1234567893","balance":5000}
2017-12-30 00:21:31.711
INFO [order-service,58b06c4c412c76cc,58b06c4c412c76cc,false]
49212 --- [nio-8090- exec-7]
p.p.s.order.controller.OrderController :
Order found:{"id":4,"status":"ACCEPTED",
"price":3752,"customerId":2,"accountId":4,"prod uctIds":[9,2]}
2017-12-30 00:21:31.722
INFO [order-service,58b06c4c412c76cc,58b06c4c412c76cc,false]
49212 --- [nio-8090- exec-7]
p.p.s.order.controller.OrderController :
Account modified:{"accountId":4,"price":3752}
2017-12-30 00:21:31.723
INFO [order-service,58b06c4c412c76cc,58b06c4c412c76cc,false]
49212 --- [nio-8090- exec-7]
p.p.s.order.controller.OrderController :
Order status changed:{"status":"DONE"}
```

### 9.4.2　使用 Kibana 搜索事件

Spring Cloud Sleuth 会自动将 HTTP 标头 X-B3-SpanId 和 X-B3-TraceId 添加到所有请求和响应中。这些字段也将作为 spanId 和 traceId 包含在 MDC 中。但在转移到查看 Kibana 仪表板之前，不妨来看一看图 9.5，这是一个顺序示意图，它说明了示例微服务之间的通信流程。

order-service 服务公开了两个可用的方法。第一个方法是创建新订单，第二个方法是确认订单。事实上，第一个 POST /方法将通过 customer-service 服务直接从 customer-service 服务、product-service 服务和 account-service 服务调用所有其他服务的端点。第二个 PUT /{id}方法仅与 account-service 服务中的一个端点集成。

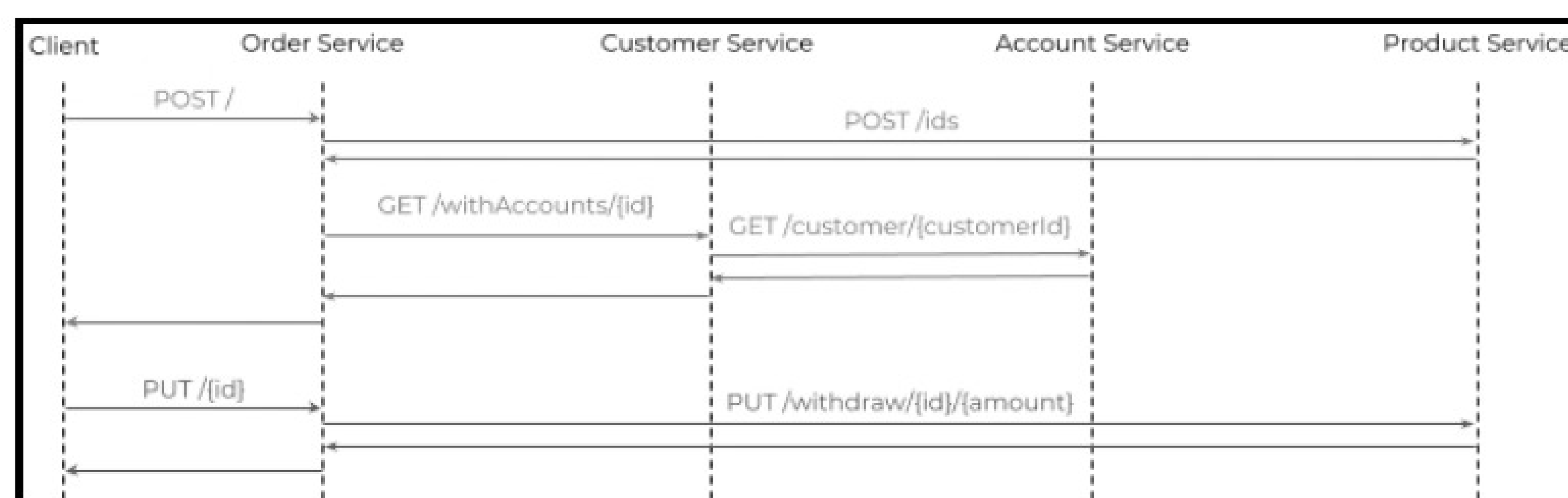



图 9.5　示例微服务之间的通信流程

现在可以通过存储在 ELK 堆栈中的日志条目来映射先前描述的流程。在使用 Kibana 作为日志聚合器的情况下，再加上由 Spring Cloud Sleuth 生成的字段，开发人员即可通过使用跟踪或跨度 ID 过滤它们来轻松查找条目。在图 9.6 所示的示例中，可以发现与 POST /方法调用的 order-service 服务端点相关的所有事件，其 X-B3-TraceId 字段等于 103ec949877519c2。

| Time | message | appName |
|---|---|---|
| January 11th 2018, 14:28:56.189 | Account found: {"id":8,"number":"1234567897","balance":3380} | order-service |
| January 11th 2018, 14:28:56.188 | Discounted price: {"price":1537} | order-service |
| January 11th 2018, 14:28:56.187 | Customer found: {"id":3,"name":"Jacob Ryan","type":"VIP","accounts":[{"id":7,"number":"1234567896","balance":0},{"id":8,"number":"1234567897","balance":3380},{"id":9,"number":"1234567898","balance":50000}]} | order-service |
| January 11th 2018, 14:28:56.183 | Accounts found: [{"id":7,"number":"1234567896","balance":0},{"id":8,"number":"1234567897","balance":3380},{"id":9,"number":"1234567898","balance":50000}] | customer-service |
| January 11th 2018, 14:28:56.162 | Products found: [{"id":8,"name":"Test8","price":1250},{"id":10,"name":"Test10","price":800}] | order-service |
| January 11th 2018, 14:28:56.162 | Products found: {"count":2} | product-service |

图 9.6　与 POST /方法调用端点相关的所有事件

图 9.7 也是一个示例，它类似于上一个示例，但处理请求期间存储的所有事件都将发送到 PUT /{id}端点。这些条目也已经被 X-B3-TraceId 字段过滤掉，X-B3-TraceId 字段的值等于 7070b90bfb36c961。

| Time | message | appName |
|---|---|---|
| December 29th 2017, 15:39:43.029 | Order status changed: {"status":"DONE"} | order-service |
| December 29th 2017, 15:39:43.029 | Account found: {"id":9,"number":"1234567898","balance":5000,"customerId":3} | account-service |
| December 29th 2017, 15:39:43.029 | Current balance: {"balance":2953} | account-service |
| December 29th 2017, 15:39:43.028 | Account modified: {"accountId":9,"price":2047} | order-service |
| December 29th 2017, 15:39:43.014 | Order found: {"id":20,"status":"ACCEPTED","price":2047,"customerId":3,"accountId":9,"productIds":[10,2]} | order-service |

图 9.7　与 POST /{id}方法调用端点相关的所有事件

在图 9.8 中可以看到完整的字段列表，这些字段已由微服务应用程序发送到 Logstash。Spring Cloud Sleuth 库已将包含 X-前缀的字段包含在消息中。

▾ December 29th 2017, 15:39:43.029 Account found: {"id":9,"number":"1234567898","balance":5000,"customerId":3}

Table　JSON

| | Field | Value |
|---|---|---|
| t | @metdata.ip_address | 192.168.99.1 |
| ⊙ | @timestamp | December 29th 2017, 15:39:43.029 |
| t | @version | 1 |
| ? | X-B3-ParentSpanId | 7079b90bfb36c961 |
| ? | X-B3-SpanId | 2db3a2acca189c51 |
| ? | X-B3-TraceId | 7079b90bfb36c961 |
| ? | X-Span-Export | false |
| t | _id | JCK2omABfCW_oGSV42bG |
| t | _index | micro-account-service |
| # | _score | - |
| t | _type | doc |
| t | appName | account-service |
| t | host | 192.168.99.1 |
| t | level | INFO |
| t | logger_name | pl.piomin.services.account.controller.AccountController |
| t | message | Account found: {"id":9,"number":"1234567898","balance":5000,"customerId":3} |
| # | port | 57,684 |
| t | thread_name | http-nio-8091-exec-2 |

图 9.8　完整的字段列表

## 9.4.3　集成 Sleuth 和 Zipkin

Zipkin 是一种流行的开源分布式跟踪系统，它有助于收集分析基于微服务的架构中的延迟问题所需的计时数据。它能够使用用户界面 Web 控制台收集、查找和可视化数据。Zipkin 用户界面提供了一个依赖关系图，显示系统中所有应用程序处理了多少个跟踪请求。Zipkin 由 4 个元素组成，前面已经提到过其中一个，即 Web 用户界面。第二个是 Zipkin 收集器，它负责验证、存储和索引所有传入的跟踪数据。Zipkin 使用 Cassandra 作为默认的后端存储。它本身也支持 Elasticsearch 和 MySQL。最后一个元素是查询服务，它提供了一个简单的 JSON API，用于查找和检索跟踪。它主要由 Web 用户界面使用。

### 1．运行 Zipkin 服务器

开发人员可以通过多种方式在本地运行 Zipkin 服务器。其中一种方法涉及使用 Docker 容器。以下命令将启动内存服务器实例。

```
docker run -d --name zipkin -p 9411: 9411 openzipkin / zipkin
```

在运行 Docker 容器之后，Zipkin API 在 http://192.168.99.100:9411 可用。或者，也可以使用 Java 库和 Spring Boot 应用程序启动它。要为应用程序启用 Zipkin，应该将以下依

赖项包含在 Maven 的 pom.xml 文件中，如以下代码片段所示。默认版本由 spring-cloud-dependencies 管理。具体到本示例应用程序，则使用了 Edgware.RELEASE Spring Cloud 版本列车。

```
<dependency>
    <groupId>io.zipkin.java</groupId>
    <artifactId>zipkin-server</artifactId>
</dependency>
<dependency>
    <groupId>io.zipkin.java</groupId>
    <artifactId>zipkin-autoconfigure-ui</artifactId>
</dependency>
```

在本示例系统中添加了一个新的 zipkin-service 模块。这很简单，唯一需要实现的是应用程序 main 类，它使用@EnableZipkinServer 进行注解。由于这个原因，Zipkin 实例将嵌入 Spring Boot 应用程序中。

```
@SpringBootApplication
@EnableZipkinServer
public class ZipkinApplication {

    public static void main(String[] args) {
        new
SpringApplicationBuilder(ZipkinApplication.class).web(true).run(args);
    }

}
```

为了在其默认端口上启动 Zipkin 实例，必须覆盖 application.yml 文件中的默认服务器端口。启动该应用程序之后，可在 http://localhost:9411 处使用 Zipkin API。

```
spring:
 application:
  name: zipkin-service

server:
 port: ${PORT:9411}
```

### 2．构建客户端应用程序

如果要在项目中同时使用 Spring Cloud Sleuth 和 Zipkin，则只需在依赖项中添加 spring-cloud-starter-zipkin 启动器即可，它将启用通过 HTTP API 与 Zipkin 的集成。如果已将 Zipkin 服务器作为 Spring Boot 应用程序内的嵌入式实例启动，则不必提供包含连接

地址的任何其他配置。如果使用 Docker 容器，则应覆盖 application.yml 中的默认 URL。

```
spring:
 zipkin:
  baseUrl: http://192.168.99.100:9411/
```

开发人员始终可以利用与服务发现的集成。如果通过@EnableDiscoveryClient 为使用嵌入式 Zipkin 服务器的应用程序启用了发现客户端，则可以将属性 spring.zipkin.locator.discovery.enabled 设置为 true。在这种情况下，即使它在默认端口下不可用，所有应用程序也可以通过已注册名称对其进行本地化。开发人员还应该使用 spring.zipkin.baseUrl 属性覆盖默认的 Zipkin 应用程序名称。

```
spring:
 zipkin:
  baseUrl: http://zipkin-service/
```

默认情况下，Spring Cloud Sleuth 仅发送一些选定的传入请求。它由属性 spring.sleuth.sampler.percentage 确定，其值必须是 0.0 和 1.0 之间的两倍。这个采样的解决方案已经实现，因为分布式系统之间交换的数据量有时非常高。Spring Cloud Sleuth 提供了可以实现的采样器接口，以控制采样算法。PercentageBasedSampler 类中提供了默认实现。如果想要跟踪应用程序交换的所有请求，只需声明 AlwaysSampler bean。它可能对测试目的有用。

```
@Bean
public Sampler defaultSampler() {
    return new AlwaysSampler();
}
```

（1）使用 Zipkin 用户界面分析数据

现在回到刚才的示例系统。如前所述，新的 zipkin-service 模块已被添加。我们还为所有微服务（包括 gateway-service 服务）启用了 Zipkin 跟踪。默认情况下，Sleuth 将采用值 spring.application.name 作为 span 的服务名称。开发人员可以使用 spring.zipkin.service.name 属性覆盖该名称。

要使用 Zipkin 成功测试我们的系统，必须启动微服务、网关、发现和 Zipkin 服务器。要生成并发送一些测试数据，可以运行由 pl.piomin.services.gateway.GatewayControllerTest 类实现的 JUnit 测试。它将通过 gateway-service 服务向 order-service 服务发送 100 条消息，这可从 http://localhost:8080/api/order/**处获得。

现在来分析 Zipkin 从所有服务中收集到的数据。可以使用其用户界面 Web 控制台轻松检查它。所有跟踪都标记有服务的名称跨度。如果条目有 5 个跨度，则表示进入系统

的请求已由 5 个不同的服务处理，如图 9.9 所示。

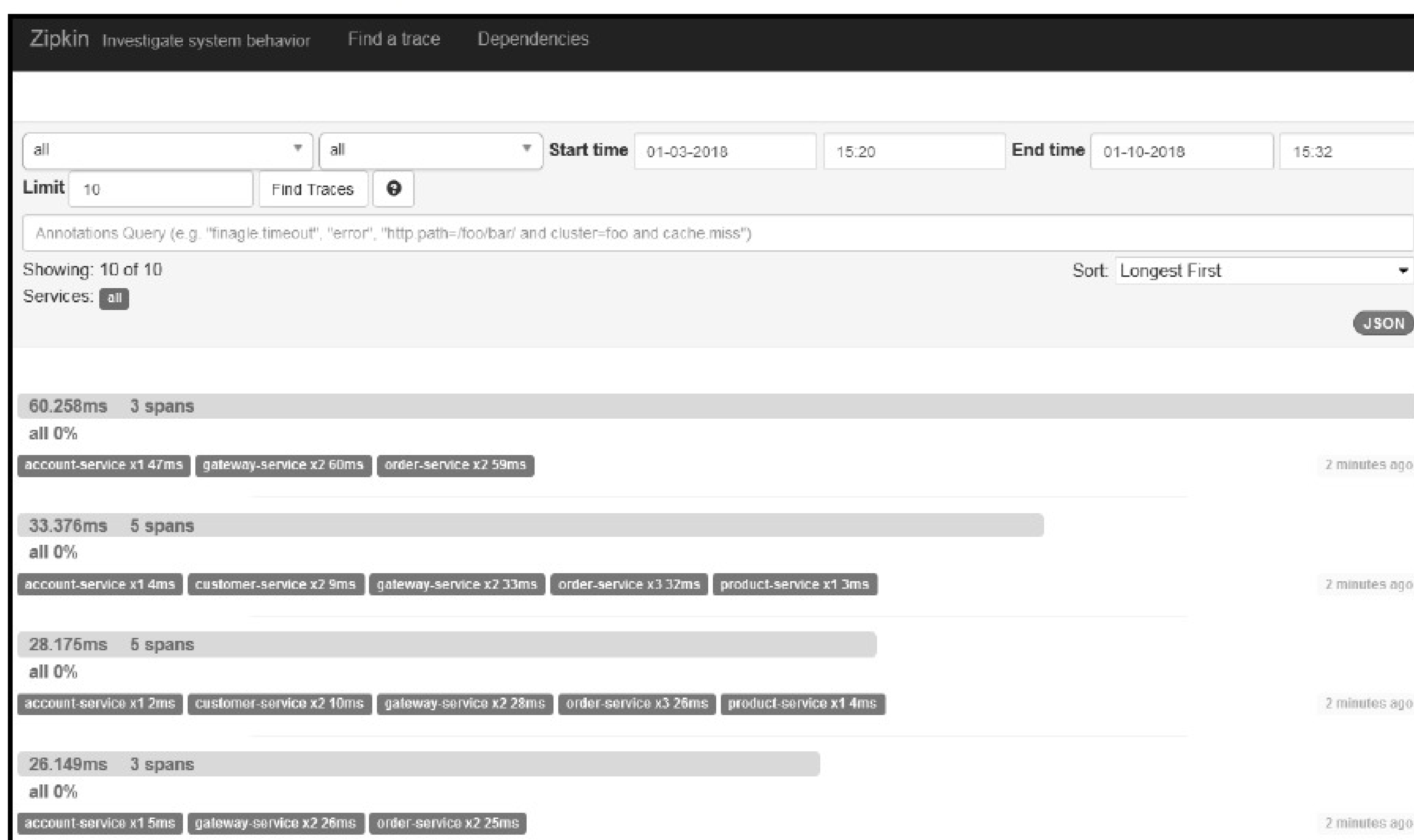



图 9.9　在 Zipkin 用户界面查看从所有服务中收集到的数据

开发人员可以使用不同的条件筛选条目，如服务名称、跨度名称、跟踪 ID、请求时间或持续时间等。Zipkin 还可以显示失败的请求，并按持续时间、降序或升序对它们进行排序，如图 9.10 所示。

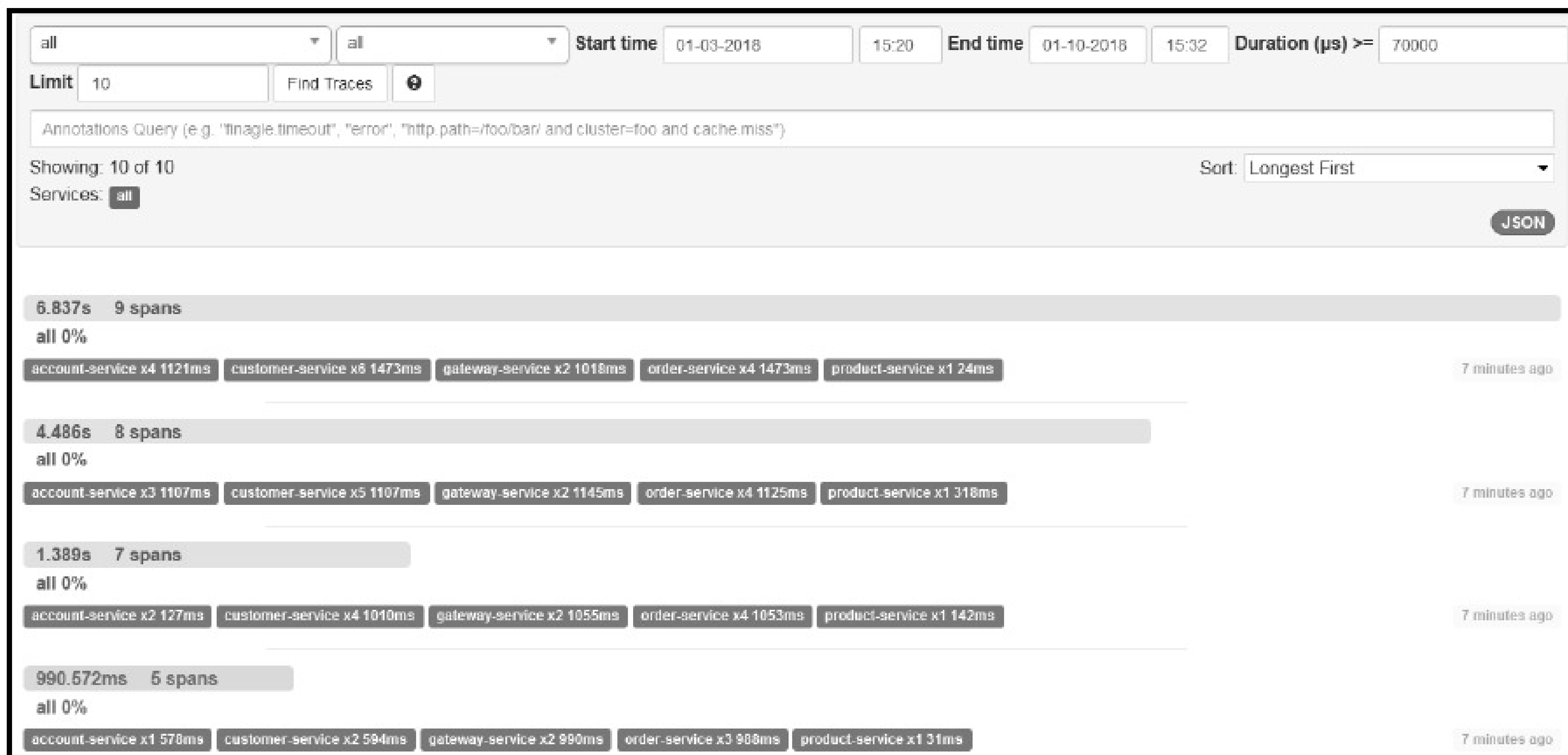



图 9.10　使用不同的条件筛选条目

开发人员还可以查看每个条目的详细信息。Zipkin 将以可视化方式显示参与通信的所有微服务之间的流程。如图 9.11 所示，它显示的是对每个传入请求的数据的计时。开发人员可以通过它了解系统中延迟的原因。

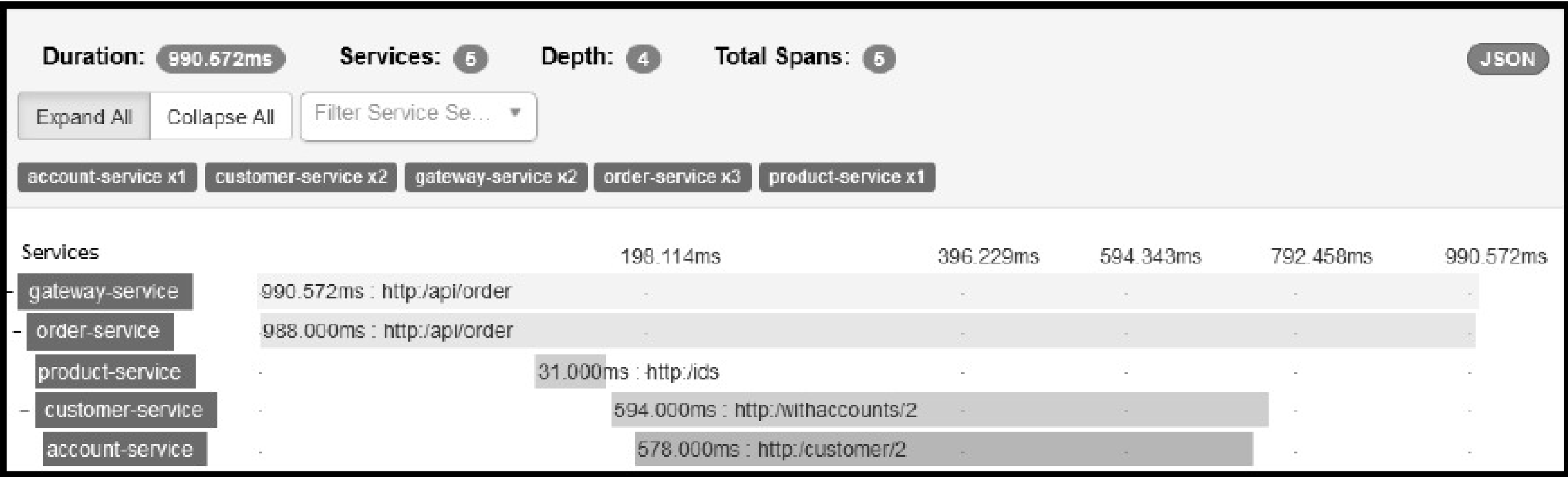



图 9.11　查看条目的详细信息

Zipkin 还提供了一些额外的有趣功能。其中之一是可视化应用程序之间的依赖关系。如图 9.12 所示，它显示了本示例系统的通信流程。

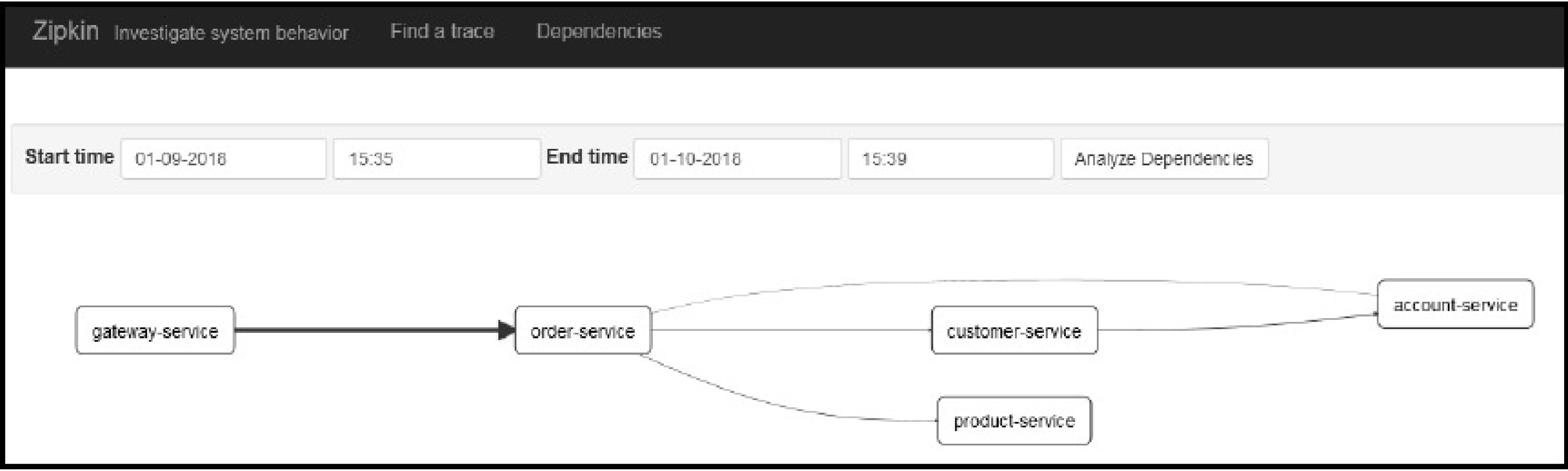



图 9.12　Zipkin 还可以将应用程序之间的依赖关系可视化

开发人员可以通过单击相关元素来查看服务之间已交换的消息数量，如图 9.13 所示。

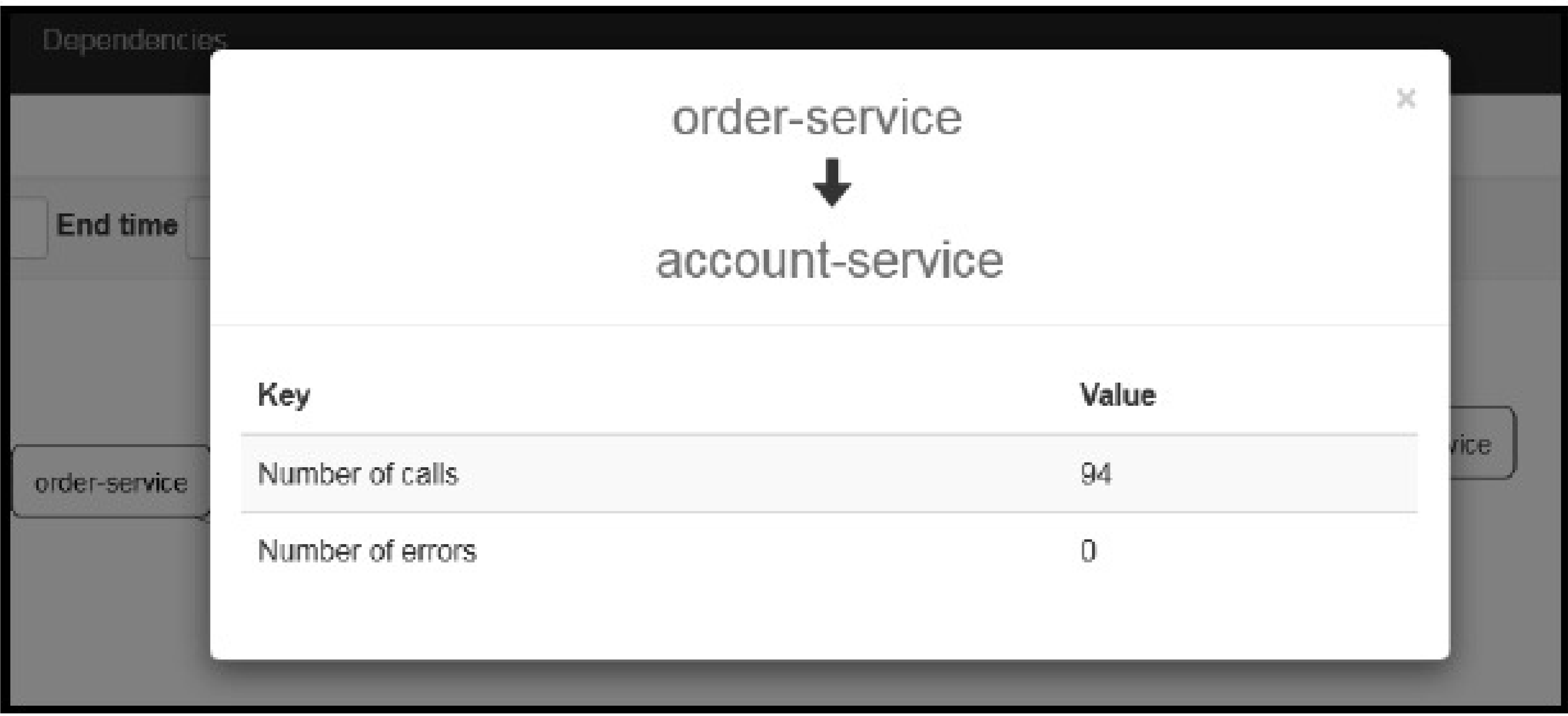



图 9.13　查看服务之间已交换的消息数量

（2）通过消息代理集成

通过 HTTP 与 Zipkin 集成不是唯一的选择。与 Spring Cloud 一样，开发人员也可以使用消息代理作为代理。有两个可用的代理——RabbitMQ 和 Kafka。第一个可以通过使用 spring-rabbit 依赖项包含在项目中，而第二个则可以使用 spring-kafka 包含在项目中。这两个代理的默认目标名称都是 zipkin。

```
<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-starter-zipkin</artifactId>
</dependency>
<dependency>
 <groupId>org.springframework.amqp</groupId>
 <artifactId>spring-rabbit</artifactId>
</dependency>
```

此功能还需要在 Zipkin 服务器端进行更改。我们已经配置了一个用户来侦听进入 RabbitMQ 或 Kafka 队列的数据，所以，要实现此目的，只需在项目中包含以下依赖项即可。开发人员仍然需要在类路径中使用 zipkin-server 和 zipkin-autoconfigure-ui 工件。

```
<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-sleuth-zipkin-stream</artifactId>
</dependency>
<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-starter-stream-rabbit</artifactId>
</dependency>
```

开发人员应该使用@EnableZipkinStreamServer 而不是@EnableZipkinServer 来注解应用程序 main 类。幸运的是，@EnableZipkinStreamServer 也是使用@EnableZipkinServer 注解的，这意味着开发人员还可以使用标准的 Zipkin 服务器端点来收集 HTTP 上的跨度，并使用 UI Web 控制台进行搜索。

```
@SpringBootApplication
@EnableZipkinStreamServer
public class ZipkinApplication {

    public static void main(String[] args) {
        new
```

```
SpringApplicationBuilder(ZipkinApplication.class).web(true).run(args);
    }

}
```

## 9.5　小　　结

在开发过程中，记录日志和跟踪通常都不是很重要，但这些是维护系统时使用的关键功能。本章将重点放在开发和运营领域，展示了如何以多种方式将 Spring Boot 微服务应用程序与 Logstash 和 Zipkin 集成。本章还提供了一些示例，以说明如何为应用程序启用 Spring Cloud Sleuth 功能，从而更轻松地监视许多微服务之间的调用。阅读完本章之后，开发人员还应该能够有效地使用 Kibana 作为日志聚合工具，并使用 Zipkin 作为跟踪工具来发现系统内部通信的“瓶颈”。

Spring Cloud Sleuth 与 Elastic Stack 和 Zipkin 一起，似乎是一个非常强大的生态系统，它提供了由许多独立微服务组成的系统监控问题的解决方案，消除了对于该问题的任何疑虑。

# 第 10 章　其他配置和发现功能

本书第 4 章“服务发现”和第 5 章“使用 Spring Cloud Config 进行分布式配置”中讨论了大量有关服务发现和分布式配置的内容。我们详细讨论了两种解决方案，其中的第一个解决方案是 Eureka，由 Netflix OSS 提供，并已被 Spring Cloud 用于服务发现；第二个解决方案是 Spring Cloud Config 项目，仅专用于分布式配置。但是，市场上有一些有趣的解决方案有效地结合了这两个功能。目前，Spring Cloud 支持其中两个。

- Consul：该产品由 HashiCorp 构建。它是一种高度可用的分布式解决方案，旨在跨动态分布式基础架构连接和配置应用程序。Consul 是一个相当复杂的产品，具有多个组件，但其主要功能是在任何基础架构中发现和配置服务。
- Zookeeper：该产品由 Apache Software Foundation 构建。它是一个用 Java 编写的分布式、分层键/值存储。它旨在维护配置信息、命名和分布式同步。与 Consul 相比，它更像是一种原始的键/值存储而不是现代服务发现工具。但是，Zookeeper 仍然非常流行，特别是对基于 Apache Software 堆栈的解决方案更是如此。

对该领域其他两种受欢迎产品的支持仍处于开发阶段。以下项目尚未添加到官方 Spring Cloud 版本列车中。

- Kubernetes：这是一个开源解决方案，旨在实现最初由 Google 创建的容器化应用程序的自动化部署、扩展和管理。这个工具现在很受欢迎。最近，Docker 平台已经开始支持 Kubernetes。
- Etcd：这是一个分布式可靠的键/值存储，用于 Go 中编写的分布式系统的最关键数据。它被许多公司和其他软件产品用于生产，如 Kubernetes。

本章将仅介绍官方支持的解决方案，即 Consul 和 Zookeeper。Kubernetes 不仅仅是一个键/值存储或服务注册表，本书将在第 14 章“Docker 支持”中详细讨论它。

## 10.1　使用 Spring Cloud Consul

Spring Cloud Consul 项目可以通过自动配置为 Consul 和 Spring Boot 应用程序提供集成。通过使用众所周知的 Spring Framework 注解样式，开发人员可以在基于微服务的环境中启用和配置常见模式。这些模式包括：使用 Consul 代理进行服务发现、使用 Consul

键/值存储进行分布式配置、使用 Spring Cloud Bus 进行分布式事件以及使用 Consul Events 等。该项目还支持基于 Netflix Ribbon 的客户端负载均衡器和基于 Netflix Zuul 的 API 网关。在开始讨论这些功能之前，必须先运行并配置 Consul 代理。

### 10.1.1 运行 Consul 代理

我们将从在本地计算机上启动 Consul 代理的最简单方法开始。可以使用 Docker 容器轻松设置独立开发模式。以下命令将从 Docker Hub 上可用的官方 Hashicorp 镜像启动 Consul 容器。

```
docker run -d --name consul -p 8500:8500 consul
```

在启动之后，Consul 的地址为 http://192.168.99.100:8500。它公开了 RESTful HTTP API，这也是它的主接口。所有 API 路由都以/V1/为前缀。当然，它不需要直接使用 API。有一些编程库可以更方便地使用 API，其中一个是 consul-api，客户端用 Java 编写，内部也由 Spring Cloud Consul 使用。Consul 提供的 Web 用户界面仪表板在与 HTTP API 相同的地址下可用，但在不同的上下文路径/ui/上。它允许查看所有已注册的服务和节点，查看所有运行状况检查及其当前状态，以及读取和设置键/值数据。

如前文所述，我们将使用 Consul 的 3 个不同功能——代理、事件和键值存储。它们的对应端点分别是/agent、/event 和/kv。最有趣的代理端点是与服务注册相关的端点。表 10.1 是这些代理端点的列表。

表 10.1　代理端点列表

| 方　法 | 路　径 | 说　明 |
|---|---|---|
| GET | /agent/services | 它将返回使用本地代理注册的服务列表。如果 Consul 以集群模式运行，则该列表可能与/catalog 端点在集群成员之间执行同步之前报告的列表不同 |
| PUT | /agent/service/register | 它将向本地代理添加新服务。代理负责管理本地服务，以及将更新发送到服务器以执行全局目录的同步 |
| PUT | /agent/service/deregister/:service_id | 它将从本地代理中删除具有 service_id 的服务。该代理负责使用全局目录取消已注册的服务 |

/kv 端点专用于管理简单的键/值存储，这对于存储服务配置或其他元数据特别有用。值得注意的是，每个数据中心都有自己的 KV 存储，因此，要在多个节点之间共享它，开发人员应该配置 Consul 复制守护进程。表 10.2 是管理键/值存储的 3 个端点的列表。

表 10.2　管理键/值存储的 3 个端点

| 方　法 | 路　径 | 说　明 |
| --- | --- | --- |
| GET | /kv/:key | 它将返回给定键名的值。如果请求的键值不存在，则返回 HTTP 状态 404 作为响应 |
| PUT | /kv/:key | 它用于向存储中添加新键值，或通过键名更新现有键值 |
| DELETE | /kv/:key | 这是最后一个 CRUD 方法，用于删除单个键值，或者删除具有相同前缀的所有键值 |

Spring Cloud 使用 Consul Events 提供动态配置重新加载。有两种简单的 API 方法。第一种是 PUT /event/fire/:name，它将触发一个新事件。第二种是 GET /event/list，它将返回一个事件列表，并且可以按名称、标记、节点或服务名称进行过滤。

## 10.1.2　在客户端集成

要在项目中激活 Consul 服务发现，应该在依赖项中包含 spring-cloud-starter-consul-discovery 启动器。如果要使用 Consul 启用分布式配置，只需包含 spring-cloud-starter-consul-config。在某些情况下，开发人员可能会在客户端应用程序中使用这两个功能，这样的话就应该声明对 spring-cloud-starter-consul-all 工件的依赖。

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-consul-all</artifactId>
</dependency>
```

默认情况下，Consul 代理应在地址 localhost:8500 下可用。如果开发人员的应用程序有所不同，则应在 application.yml 或 bootstrap.yml 文件中提供相应的地址。

```
spring:
 cloud:
  consul:
   host: 192.168.99.100
   port: 18500
```

## 10.1.3　服务发现

通过使用通用 Spring Cloud @EnableDiscoveryClient 注解 main 类，可以为应用程序启用使用 Consul 的服务发现。有关详细设置，可以参考本书第 4 章“服务发现”，因为它与 Eureka 相比没有区别，其默认服务名称也取自${spring.application.name}属性。

使用 Consul 作为发现服务器的示例微服务可以在 GitHub 存储库的 https://github.com/piomin/sample-spring-cloud-consul.git 上获得。该系统的架构与前面章节中的示例相同。它有 4 个微服务：order-service 服务、product-service 服务、customer-service 服务和 account-service 服务，API 网关在 gateway-service 模块中实现。对于服务间通信，则将 Feign 客户端与 Ribbon 负载均衡器一起使用。

```
@SpringBootApplication
@EnableDiscoveryClient
@EnableFeignClients
public class CustomerApplication {

    public static void main(String[] args) {
        new
SpringApplicationBuilder(CustomerApplication.class).web(true).run(args);
    }

}
```

默认情况下，Spring Boot 应用程序在 Consul 中注册，其实例 ID 是通过从属性 spring.application.name、spring.profiles.active 和 server.port 获取的值连接在一起而生成的。在大多数情况下，确保 ID 是唯一的就足够了，但如果需要自定义模式，则可以使用 spring.cloud.consul.discovery.instanceId 属性轻松设置。

```
spring:
 cloud:
  consul:
   discovery:
    instanceId:
${spring.application.name}:${vcap.application.instance_id:${spring.applicat
ion.instance_id:${random.value}}}
```

在启动所有示例微服务之后，即可查看 Consul 用户界面仪表板。此时应该看到注册了 4 种不同的服务，如下面的屏幕截图 10.1 所示。

或者，开发人员也可以使用 RESTful HTTP API 端点 GET /v1/agent/services 查看已注册服务的列表。以下是 JSON 响应的片段。

```
"customer-service-zone1-8092": {
 "ID": "customer-service-zone1-8092",
 "Service": "customer-service",
 "Tags": [],
 "Address": "minkowp-l.p4.org",
```

```
 "Port": 8092,
 "EnableTagOverride": false,
 "CreateIndex": 0,
 "ModifyIndex": 0
},
"order-service-zone1-8090": {
 "ID": "order-service-zone1-8090",
 "Service": "order-service",
 "Tags": [],
 "Address": "minkowp-l.p4.org",
 "Port": 8090,
 "EnableTagOverride": false,
 "CreateIndex": 0,
 "ModifyIndex": 0
}
```

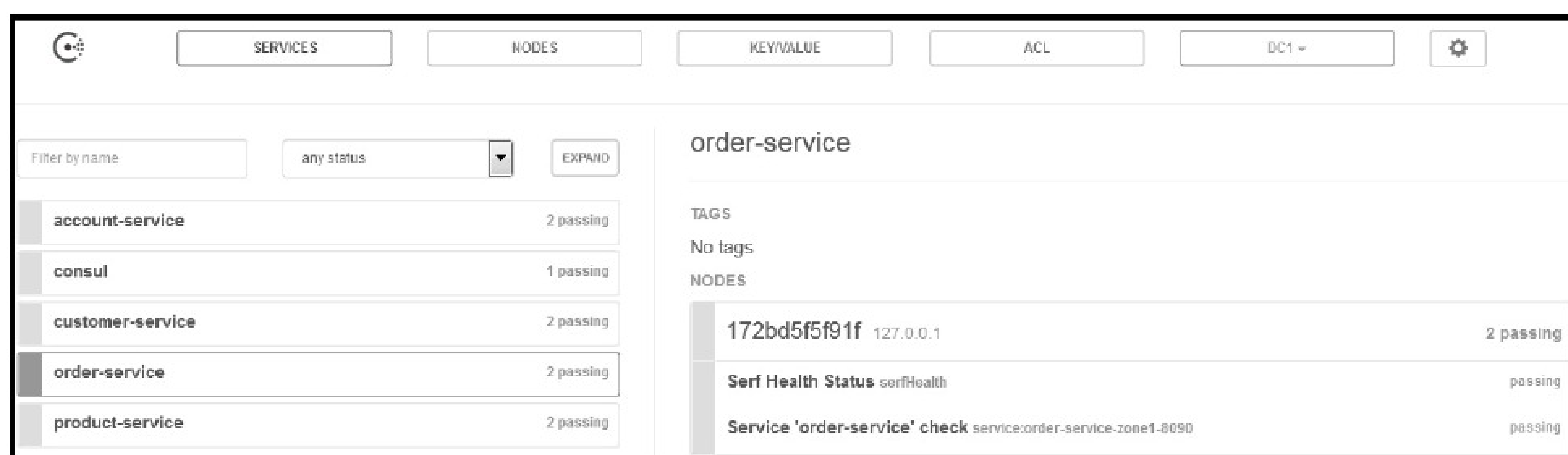



图 10.1　查看 Consul 用户界面仪表板

现在，可以使用 pl.piomin.services.order.OrderControllerTest JUnit 测试类通过向 order-service 服务发送一些测试请求来轻松测试整个系统。一切都应该工作得很好，和 Eureka 的发现一样。

## 1．运行状况检查

Consul 可以通过调用 /health 端点来检查每个已注册实例的运行状况。如果不希望在类路径中提供 Spring Boot Actuator 库，或者如果开发人员的服务存在一些问题，那么它将在 Web 仪表板上显示，如图 10.2 所示。

如果运行状况检查端点由于任何原因在不同的上下文路径下可用，则可以使用 spring.cloud.consul.discovery.healthCheckPath 属性覆盖该路径。还可以通过使用模式定义 healthCheckInterval 来更改状态刷新间隔，例如，10s 表示 10 秒，2m 表示 2 分钟。

```
spring:
 cloud:
  consul:
   discovery:
    healthCheckPath: admin/health
    healthCheckInterval: 20s
```

图 10.2　运行状况检查

### 2. 区域

本书在第 4 章“服务发现”中已经详细介绍了可用于 Eureka 发现的分区机制。当主机放置在不同的位置，而开发人员又希望在同一区域中注册的实例之间进行通信时，它非常有用。虽然 Spring Cloud Consul 的官方说明文档（http://cloud.spring.io/spring-cloud-static/spring-cloud-consul/1.2.3.RELEASE/single/spring-cloud-consul.html）对这样的解决方案未置一词，但幸运的是这并不意味着它无法实现。Spring Cloud 可以提供基于 Consul 标记的分区机制。开发人员可以使用 spring.cloud.consul.discovery.instanceZone 属性配置应用程序的默认区域。它使用传递的值设置 spring.cloud.consul.discovery.defaultZoneMetadataName 属性中配置的标记。默认元数据标签的名称为 zone。

现在继续讨论示例应用程序。我们已经使用两个配置文件（zone1 和 zone2）扩展了所有配置文件。以下是 order-service 服务的 bootstrap.yml 文件。

```
spring:
 application:
  name: order-service
 cloud:
  consul:
   host: 192.168.99.100
   port: 8500
```

```
---
spring:
 profiles: zone1
 cloud:
  consul:
   discovery:
    instanceZone: zone1
server:
 port: ${PORT:8090}

---
spring:
 profiles: zone2
  cloud:
   consul:
    discovery:
     instanceZone: zone2
server:
 port: ${PORT:9090}
```

在两个不同区域中注册的每个微服务都有两个运行实例。使用 mvn clean install 命令构建整个项目后，应该启动具有活动配置文件 zone1 或 zone2 的 Spring Boot 应用程序，如 java -jar --spring.profiles.active=zone1 target/order-service-1.0-SNAPSHOT.jar。开发人员可以在 NODES（节点）中看到使用区域标记的已注册实例的完整列表。如图 10.3 所示为此时的 Consul 仪表板视图。

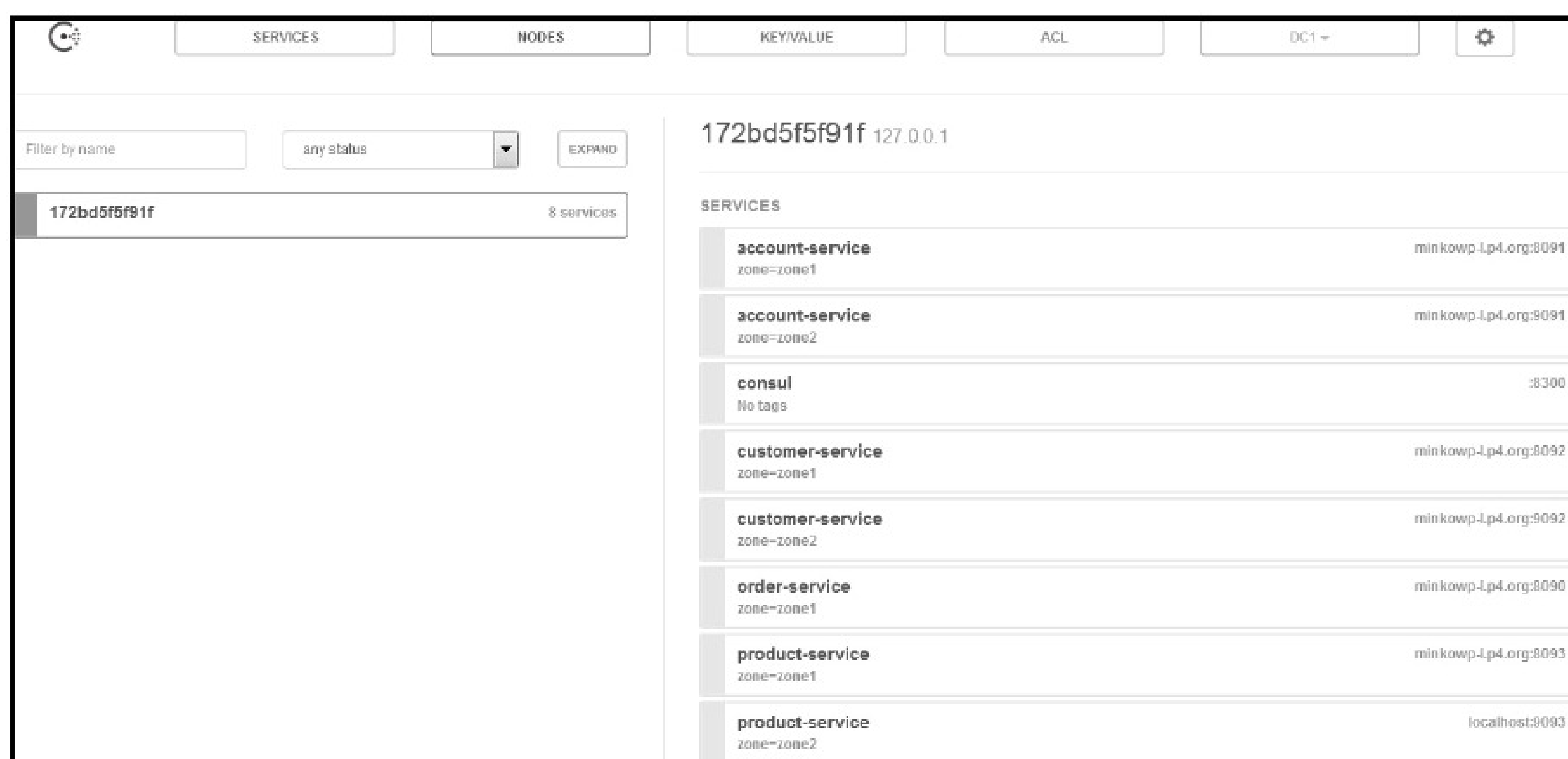



图 10.3　在 NODES 中可以看到使用区域标记的已注册实例的完整列表

本架构中的最后一个元素是基于 Zuul 的 API 网关。我们还将在不同的区域中运行两个 gateway-service 服务实例。我们希望在 Consul 中省略注册，并且只允许获取配置，Ribbon 客户端在执行负载均衡时使用该配置。以下是 gateway-service 服务的 bootstrap.yml 文件的片段。它已经通过将 spring.cloud.consul.discovery.register 和 spring.cloud.consul.discovery.registerHealthCheck 属性设置为 false 禁用了注册。

```
---
spring:
 profiles: zone1
 cloud:
 consul:
 discovery:
 instanceZone: zone1
 register: false
 registerHealthCheck: false
server:
 port: ${PORT:8080}

---
spring:
 profiles: zone2
 cloud:
 consul:
 discovery:
 instanceZone: zone2
 register: false
 registerHealthCheck: false
server:
 port: ${PORT:9080}
```

### 3. 客户端自定义设置

可以通过配置文件中的属性自定义 Spring Cloud Consul 客户端。其中一些设置已在本章前面的小节中介绍过。其他有用的设置已在表 10.3 中列出。所有这些都以 spring.cloud.consul.discovery 为前缀。

表 10.3　客户端自定义设置

| 属　性 | 默 认 值 | 说　明 |
| --- | --- | --- |
| enabled | true | 设置是否为应用程序启用或禁用 Consul 发现 |
| failFast | true | 在服务注册期间如果为 true 则抛出异常；否则，它会记录警告日志 |

续表

| 属　性 | 默 认 值 | 说　明 |
| --- | --- | --- |
| hostname | - | 在 Consul 中注册时设置实例的主机名 |
| preferIpAddress | false | 强制应用程序在注册期间发送其 IP 地址而不是主机名 |
| scheme | http | 设置服务是否在 HTTP 或 HTTPS 协议下可用 |
| serverListQueryTags | - | 允许通过单个标记过滤服务 |
| ServiceName | - | 覆盖服务名称，默认情况下从属性 spring.application.name 获取 |
| tags | - | 使用在注册服务时使用的值设置列表标记 |

### 4．以集群模式运行

到目前为止，我们通常会启动一个单独的 Consul 实例。但这只是开发模式中的合适解决方案，对于生产模式来说是不够的。在生产模式中，我们会希望拥有一个可扩展的生产级服务发现基础架构，其中包含一些在集群内部协同工作的节点。Consul 提供了对集群的支持，它将在成员之间通信时使用 Gossip 协议，在选举领导时使用 Raft 共识协议。在此我们不想介绍该过程的细节，但是应该说明一些关于 Consul 架构的基础知识。

我们已经讨论过 Consul 代理，但它究竟是什么，以及它的作用是什么并没有得到解释。代理是 Consul 集群的每个成员长期运行的守护程序。它可以在客户端或服务器模式下运行。所有代理都负责在全局范围内运行检查并保持在不同节点和同步中注册的服务。

本节的主要目标是使用 Docker 镜像设置和配置 Consul 集群。首先，我们将启动容器，它充当集群的领导者。当前使用的 Docker 命令与独立的 Consul 服务器只有一个区别。我们设置了环境变量 CONSUL_BIND_INTERFACE=eth0，以便将集群代理的网络地址从 127.0.0.1 更改为可用于其他成员容器的网络地址。笔者的 Consul 服务器现在在内部地址 172.17.0.2 上运行。要查看你的地址（它应该是相同的），可以运行命令 docker logs consul。容器启动后将记录适当的信息。

```
docker run -d --name consul-1 -p 8500:8500 -e CONSUL_BIND_INTERFACE=eth0
consul
```

了解该地址非常重要，因为现在我们必须将它作为集群连接参数传递给每个成员容器启动命令。我们还通过将 0.0.0.0 设置为客户端地址将其绑定到所有接口。现在，可以使用-p 参数轻松地在客户端代理 API 之外公开客户端代理 API。

```
docker run -d --name consul-2 -p 8501:8500 consul agent -server -
client=0.0.0.0 -join=172.17.0.2
docker run -d --name consul-3 -p 8502:8500 consul agent -server -
client=0.0.0.0 -join=172.17.0.2
```

在使用 Consul 代理运行两个容器之后，可以通过在领导者（Leader）的容器上执行

如图 10.4 所示的命令来检查集群成员的完整列表。

```
$ docker exec -t consul-1 consul members
Node          Address          Status  Type    Build  Protocol  DC   Segment
4b3c3c84dd96  172.17.0.3:8301  alive   server  0.9.3  2         dc1  <all>
7b4c661849ed  172.17.0.2:8301  alive   server  0.9.3  2         dc1  <all>
8429a8226624  172.17.0.4:8301  alive   server  0.9.3  2         dc1  <all>
```

图 10.4　使用命令检查集群成员的完整列表

Consul 服务器代理在 8500 端口上公开，而成员代理在端口 8501 和 8502 上公开。即使微服务实例将自身注册到成员代理，集群的所有成员也可以看到它，如图 10.5 所示。

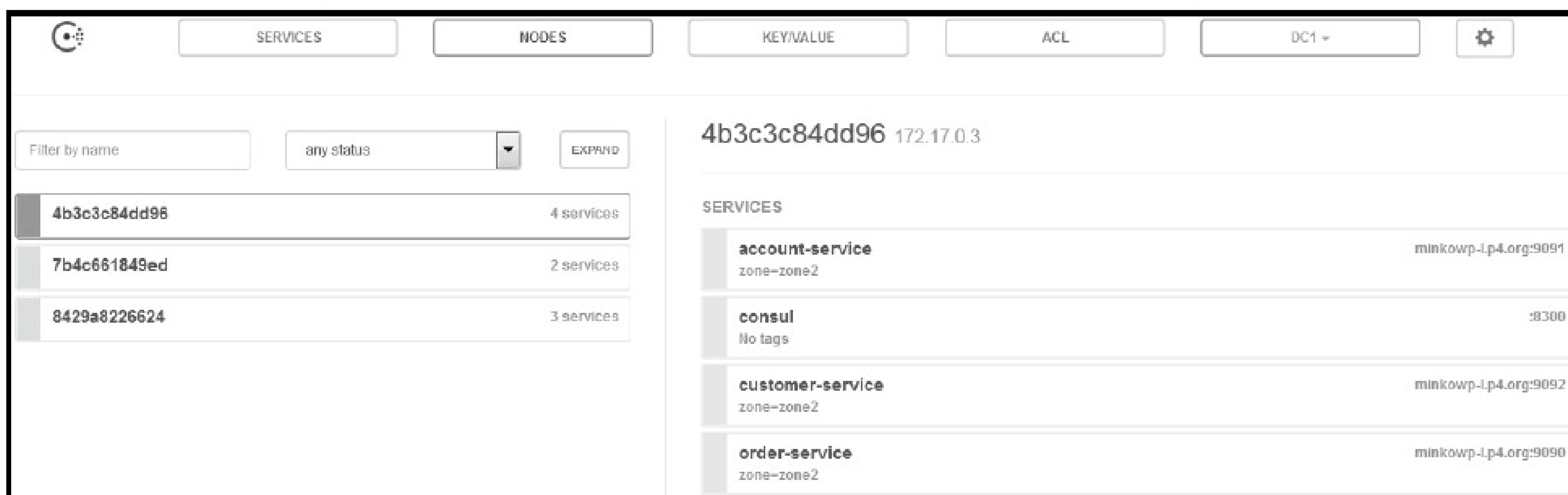



图 10.5　集群的所有成员都能看到在成员代理上注册的微服务实例

开发人员可以通过更改配置属性轻松更改 Spring Boot 应用程序的默认 Consul 代理地址。

```
spring:
 application:
  name: customer-service
 cloud:
  consul:
   host: 192.168.99.100
   port: 8501
```

## 10.1.4　分布式配置

对于在类路径中使用 Spring Cloud Consul Config 库的应用程序来说，它们将在引导阶段从 Consul 键/值存储中获取配置。也就是说，默认情况下，会存储在/config 文件夹中。当开发人员要创建一个新键值时，必须设置一个文件夹路径。该路径将用于标识键值并将其分配给应用程序。Spring Cloud Config 会尝试根据应用程序名称和活动配置文件解析存

储在文件夹中的属性。假设我们已经在 bootstrap.yml 文件中将 spring.application.name 属性设置为 order-service，并将 spring.profiles.active 运行参数设置为 zone1，那么它会尝试按以下顺序查找属性源：config/order-service、zone1/、config/order-service/、config/ application、zone1/、config/application/。对于所有没有与服务相关的属性源的应用程序来说，包含前缀 config/application 的所有文件夹就是它们的默认配置。

### 1．管理 Consul 中的属性

向 Consul 添加单个键值的最便捷方式是通过其 Web 仪表板。另一种方法是使用/kv HTTP 端点，这在本章开头已经介绍过。在使用 Web 控制台时，必须转到 KEY/VALUE（键/值）视图，然后就可以查看所有当前存在的键和值，并通过以任何格式提供其完整路径和值来创建新的键，如图 10.6 所示。

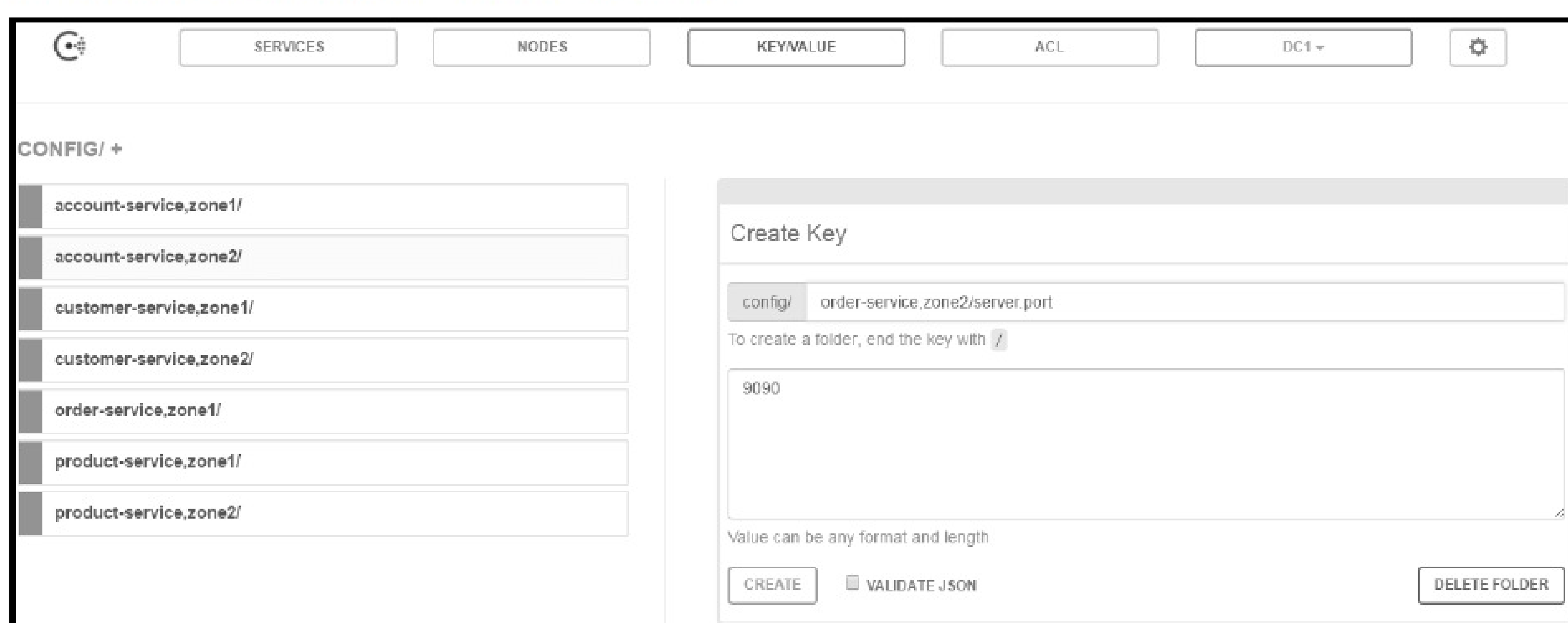



图 10.6　查看键和值

每个键值都可以更新或删除，如图 10.7 所示。

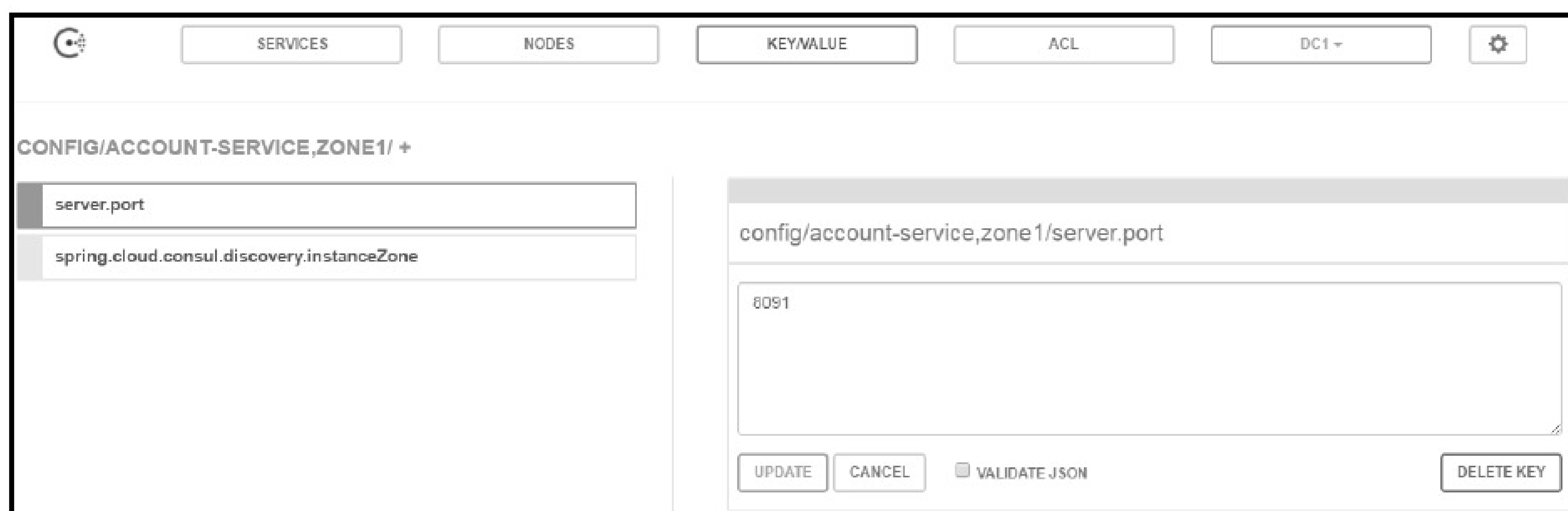



图 10.7　键值管理

要访问使用 Consul 中存储的属性源的示例应用程序，应该切换到与上一个示例相同的存储库中的分支配置。我们已经为每个微服务创建了键值 server.port 和 spring.cloud.consul.discovery.instanceZone，而不是在 application.yml 或 bootstrap.yml 文件中定义它。

### 2. 客户端定制

可以使用以下属性自定义 Consul Config 客户端，这些属性以 spring.cloud.consul.config 为前缀。

- enabled：通过将此属性设置为 false，可以禁用 Consul Config。如果已经包含了 spring-cloud-starter-consul-all，那么这将非常有用，因为后者会启用发现和分布式配置。
- fail-fast：设置是否在配置查找期间抛出异常或在连接失败时抛出日志警告。将其设置为 true 允许应用程序正常继续启动。
- prefix：这将设置所有配置值的基本文件夹，默认情况下为/config。
- defaultContext：这将设置所有没有特定配置的应用程序使用的文件夹名称，默认情况下为/application。例如，如果将其覆盖到 app，则应在/config/apps 文件夹中搜索属性。
- profileSeparator：默认情况下，使用逗号将应用程序名称分隔为配置文件。该属性允许覆盖该分隔符的值。例如，如果将其设置为::，则应创建文件夹/config/order-service::zone1/。以下就是一个示例。

```
spring:
 cloud:
  consul:
   config:
    enabled: true
    prefix: props
    defaultContext: app
    profileSeparator: '::'
```

有时，开发人员会希望存储以 YAML 或 Properties 格式创建的一堆属性，而不是单个键/值对。在这种情况下，应该将 spring.cloud.consul.config.format 属性设置为 YAML 或 PROPERTIES。然后，应用程序将查找位于具有数据键的文件夹内的配置属性，如 config/order-service、zone1/data、config/order-service/data、config/application、zone1/data 或 config/application/data。可以使用 spring.cloud.consul.config.data-key 属性更改默认数据键。

### 3. 观察配置更改

10.1.3 节中讨论的示例将在应用程序启动时加载配置。如果开发人员希望重新加载该

配置，则应将 HTTP POST 发送到 /refresh 端点。为了检查这样的刷新对示例应用程序的影响方式，我们修改了负责创建一些测试数据的应用程序代码片段。到目前为止，它已作为存储库@Bean 提供，其中包含一些硬编码的内存中的对象。来看以下代码。

```
@Bean
CustomerRepository repository() {
    CustomerRepository repository = new CustomerRepository();
    repository.add(new Customer("John Scott", CustomerType.NEW));
    repository.add(new Customer("Adam Smith", CustomerType.REGULAR));
    repository.add(new Customer("Jacob Ryan", CustomerType.VIP));
    return repository;
}
```

我们的目标是使用 Consul 键/值功能将上述代码移动到配置存储。要完成此目标，必须为每个对象创建 3 个键，其名称分别为 id、name 和 type。配置则是从具有 repository 前缀的属性加载的。

```
@RefreshScope
@Repository
@ConfigurationProperties(prefix = "repository")
public class CustomerRepository {

    private List<Customer> customers = new ArrayList<>();

    public List<Customer> getCustomers() {
        return customers;
    }

    public void setCustomers(List<Customer> customers) {
        this.customers = customers;
    }
    // ...
}
```

下一步是使用 Consul Web 仪表板为每个服务定义适当的键值。如图 10.8 所示的是包含 Customer 对象的列表的示例配置。该列表将在应用程序启动时初始化。

开发人员可以更改每个属性的值。由于 Consul 能够查看键值前缀，更新事件将自动发送到应用程序。如果存在新配置数据，则刷新事件将发布到队列。所有队列和交换消息都是在 Spring Cloud Bus 的应用程序启动时创建的，它作为 spring-cloud-starter-consul-all 的依赖项包含在项目中。如果应用程序接收到此类事件，那么它会在日志中打印以下

信息。

```
Refresh keys changed: [repository.customers[1].name]
```

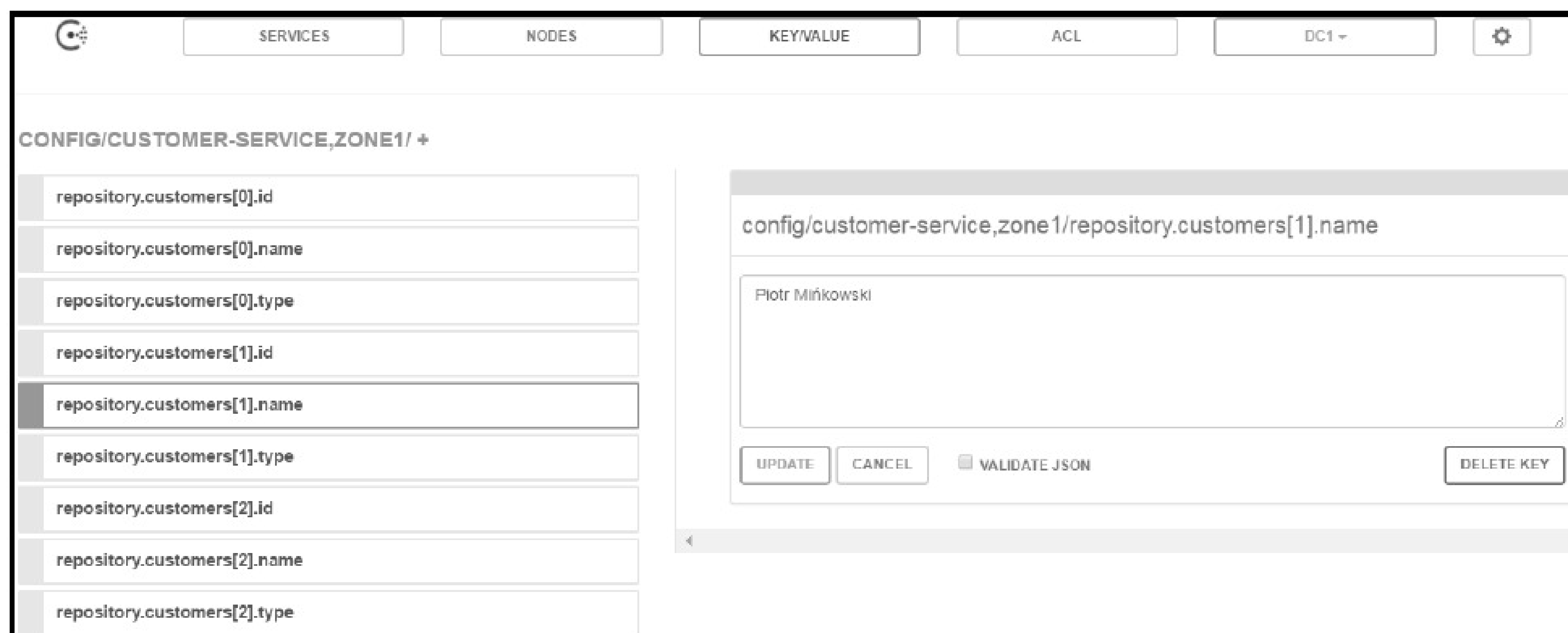



图 10.8　包含 Customer 对象的列表的示例配置

## 10.2　使用 Spring Cloud Zookeeper

Spring Cloud 支持作为微服务架构一部分的各种产品。例如，在阅读本章过程中，开发人员可以将 Consul 与作为发现工具的 Eureka 进行比较，也可以将 Consul 与作为分布式配置工具的 Spring Cloud Config 进行比较，通过这种比较，可以对 Spring Cloud 所支持产品的丰富性有更加深刻的认识。Zookeeper 是另一种解决方案，它可以作为之前列出的产品的替代选择。与 Consul 一样，它可用于服务发现和分布式配置。要在项目中启用 Spring Cloud Zookeeper，应该包含用于服务发现功能的 spring-cloud-starter-zookeeper-discovery 启动器，或用于配置服务器功能的 spring-cloud-starter-zookeeper-config 启动器。

或者，开发人员也可以声明一个 spring-cloud-starter-zookeeper-all 依赖项，它可以激活应用程序的所有功能。当然，也不要忘记包含 spring-boot-starter-web，因为它仍然需要提供 Web 功能。

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-zookeeper-all</artifactId>
</dependency>
<dependency>
```

```
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

Zookeeper 连接设置是自动配置的。默认情况下，客户端会尝试连接到 localhost:2181。要覆盖它，应该使用当前服务器网络地址定义 spring.cloud.zookeeper.connect-string 属性。

```
spring:
 cloud:
  zookeeper:
   connect-string: 192.168.99.100:2181
```

与 Spring Cloud Consul 一样，Zookeeper 支持 Spring Cloud Netflix 提供的所有最流行的通信库，如 Feign、Ribbon、Zuul 或 Hystrix。在开始处理示例实现之前，开发人员必须先启动 Zookeeper 实例。

### 10.2.1　运行 Zookeeper

为简便起见，可以使用 Docker 镜像在本地计算机上启动 Zookeeper。以下命令将启动 Zookeeper 服务器实例。由于它有快速失败（Fails Fast）机制，所以最好的方法是始终重新启动它。

```
docker run -d --name zookeeper --restart always -p 2181:2181 zookeeper
```

与先前讨论的在此领域的解决方案（如 Consul 或 Eureka）相比，Zookeeper 没有提供简单的方便开发人员管理的 RESTful API 或 Web 管理控制台。它有一个用于 Java 和 C 语言的官方 API 绑定。开发人员也可以使用它的命令行接口，它可以在 Docker 容器中轻松启动。以下命令将启动带有命令行客户端的容器，并可将其链接到 Zookeeper 服务器容器。

```
docker run -it --rm --link zookeeper:zookeeper zookeeper zkCli.sh -server
zookeeper
```

Zookeeper CLI 允许执行一些有用的操作，如下所示。

- 创建 znode：要使用给定路径创建 znode，可以使用命令 create /path /data。
- 获取数据：命令 get /path 将返回与 znode 关联的数据和元数据。
- 观察更改的 znode：如果 znode 或 znode 的子数据发生更改，则显示通知。观察只能使用 get 命令设置。
- 设置数据：要设置 znode 数据，可以使用命令 set /path /data。

- 创建 znode 的子代：此命令与用于创建单个 znode 的命令类似。唯一的区别是子 znode 的路径将包括父路径。其命令格式为 create /parent /path /subnode /path /data。
- 列出 znode 的子节点：可以使用 ls /path 命令显示它。
- 检查状态：可以使用 stat /path 命令检查。状态将描述指定 znode 的元数据，如时间戳或版本号。
- 删除/删除 znode：rmr /path 命令可以删除 znode 及其所有子节点。

请注意，术语 Zookeeper 节点（znode）在这里是首次出现。在存储数据时，Zookeeper 将使用树结构，其中每个节点称为 znode。这些 znode 的名称基于从根节点获取的路径。每个节点都有一个名称。可以使用从根节点开始的绝对路径访问它。此概念类似于 Consul 文件夹，并已用于在键/值存储中创建键。

## 10.2.2　服务发现

Apache Zookeeper 最流行的 Java 客户端库是 Apache Curator。它提供了一个 API 框架和实用程序，使 Apache Zookeeper 的应用变得更加容易。它还包括常见用例和扩展，如服务发现或 Java 8 异步 DSL。Spring Cloud Zookeeper 可以利用一个这样的扩展来实现服务发现。Spring Cloud Zookeeper 对 Curator 库的使用对于开发人员来说是完全透明的，所以在这里就不必做更多的介绍。

### 1．客户端实现

客户端的用法与其他服务发现相关的 Spring Cloud 项目相同。应用程序的 main 类或 @Configuration 类应使用@EnableDiscoveryClient 注解。默认的服务名称、实例 ID 和端口分别取自 spring.application.name、Spring Context ID 和 server.port。示例应用程序源代码位于 GitHub 存储库（https://github.com/piomin/sample-spring-cloud-zookeeper.git）中。从根本上说，除了 Spring Cloud Zookeeper Discovery 依赖项之外，它与为 Consul 引入示例系统没有什么不同。它仍然由 4 个微服务组成，这些微服务之间可以相互通信。现在，在克隆存储库之后，可以使用 mvn clean install 命令构建它。然后使用 java -jar 命令运行具有活动配置文件名称的每个服务，如 java -jar -spring.profiles.active=zone1 order-service/target/order-service-1.0-SNAPSHOT.jar。

可以使用 CLI 命令 ls 和 get 查看已注册服务和实例的列表。默认情况下，Spring Cloud Zookeeper 会注册/services 根文件夹中的所有实例。它可能会被 spring.cloud.zookeeper.discovery.root 属性覆盖，如图 10.9 所示。

```
[zk: zookeeper(CONNECTED) 13] ls /services
[product-service, order-service, account-service, customer-service]
[zk: zookeeper(CONNECTED) 14] ls /services/order-service
[987ae9bd-6e80-41ad-899d-1ab68709717b, 1c5dcd1a-423c-487f-a006-de3d1caee224]
[zk: zookeeper(CONNECTED) 15] get /services/order-service/987ae9bd-6e80-41ad-899d-1ab68709717b
{"name":"order-service","id":"987ae9bd-6e80-41ad-899d-1ab68709717b","address":"minkowp-1.p4.org","port":9090,"sslPort":null,"payload":{"@class":"org.springframework.cloud.zookeeper
.discovery.ZookeeperInstance","id":"order-service:zone2:9090","name":"order-service","metadata":{"instance_status":"UP"}},"registrationTimeUTC":1515075800204,"serviceType":"DYNAMIC
","uriSpec":{"parts":[{"value":"scheme","variable":true},{"value":"://","variable":false},{"value":"address","variable":true},{"value":":","variable":false},{"value":"port","variab
le":true}]}}
cZxid = 0xb
ctime = Thu Jan 04 14:23:20 GMT 2018
mZxid = 0xb
mtime = Thu Jan 04 14:23:20 GMT 2018
pZxid = 0xb
cversion = 0
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x10000081d640002
dataLength = 552
numChildren = 0
```

图 10.9　使用带命令行客户端的 Docker 容器检查当前已注册服务的列表

### 2．Zookeeper 依赖项

Spring Cloud Zookeeper 还有一个名为 Zookeeper 依赖项（Zookeeper Dependencies）的附加功能。这里的依赖项应理解为在 Zookeeper 中注册的其他应用程序，这些应用程序通过 Feign 客户端或 Spring RestTemplate 调用。可以将这些依赖项作为应用程序的属性提供。在将 spring-cloud-starter-zookeeper-discovery 启动器包含到项目中之后，可以通过自动配置启用该功能。当然，也可以通过将 spring.cloud.zookeeper.dependency.enabled 属性设置为 false 来禁用它。

Zookeeper 依赖项机制的配置随着 spring.cloud.zookeeper.dependencies.*属性一起提供。以下是来自 order-service 服务的 bootstrap.yml 文件的片段。此服务可与所有其他可用服务集成。

```
spring:
 application:
  name: order-service
 cloud:
  zookeeper:
   connect-string: 192.168.99.100:2181
  dependency:
   resttemplate:
    enabled: false
  dependencies:
   account:
    path: account-service
    loadBalancerType: ROUND_ROBIN
    required: true
   customer:
    path: customer-service
    loadBalancerType: ROUND_ROBIN
    required: true
   product:
```

```
    path: product-service
    loadBalancerType: ROUND_ROBIN
    required: true
```

现在来仔细看一看前面的配置。每个被调用服务的root属性是别名，然后可以由Feign客户端或@LoadBalanced RestTemplate 用作服务名称。

```
@FeignClient(name = "customer")
public interface CustomerClient {

    @GetMapping("/withAccounts/{customerId}")
    Customer findByIdWithAccounts(@PathVariable("customerId") Long
customerId);

}
```

配置中的下一个非常重要的字段是路径。它设置在 Zookeeper 中注册依赖项的路径。因此，如果该属性具有的值为 customer-service，则意味着 Spring Cloud Zookeeper 会尝试在路径/services/customer-service 下查找相应的服务 znode。还有一些其他属性可以自定义客户端的行为。其中之一是 loadBalancerType，用于应用负载均衡策略。开发人员可以在 3 种可用策略之间选择——ROUND_ROBIN、RANDOM 和 STICKY。还可以为每个服务映射将 required 属性设置为 true。现在，如果应用程序在引导期间无法检测到所需的依赖项，则无法启动。Spring Cloud Zookeeper 依赖项还允许管理 API 版本（属性 contentTypeTemplate 和 versions）和请求标头（headers 属性）。

默认情况下，Spring Cloud Zookeeper 允许 RestTemplate 与依赖项进行通信。在分支依赖项（https://github.com/piomin/sample-spring-cloud-zookeeper/tree/dependencies）提供的示例应用程序中，我们使用了 Feign 客户端而不是@LoadBalanced RestTemplate。为了禁用该功能，应该将属性 spring.cloud.zookeeper.dependency.resttemplate.enabled 设置为 false。

### 10.2.3　分布式配置

Zookeeper 的配置管理与 Spring Cloud Consul Config 的配置管理非常相似。默认情况下，所有属性源都存储在 /config 文件夹（或 Zookeeper 术语中的 znode）中。如前文所述，假设在 bootstrap.yml 文件中将 spring.application.name 属性设置为 order-service，并将 spring.profiles.active 运行参数设置为 zone1，那么它会尝试按以下顺序查找属性源：config/order-service、zone1/、config/order-service/、config/application、zone1/、config/application/。存储在命名空间中具有 config/application 前缀的文件夹中的属性可用于使用

Zookeeper 进行分布式配置的所有应用程序。

要访问示例应用程序，需要切换到 https://github.com/piomin/sample-spring-cloud-zookeeper.git 存储库中的分支配置。本地 application.yml 或 bootstrap.yml 文件中定义的配置如下所示，现已移至 Zookeeper。

```
---
spring:
 profiles: zone1
server:
 port: ${PORT:8090}

---
spring:
 profiles: zone2
server:
 port: ${PORT:9090}
```

必须使用 CLI 创建所需的 znode。如图 10.10 所示的是使用给定路径创建 znode 的 Zookeeper 命令列表，这里使用了 create /path /data 命令。

```
[zk: zookeeper(CONNECTED) 11] create /config ""
Created /config
[zk: zookeeper(CONNECTED) 12] create /config/order-service,zone1 ""
Created /config/order-service,zone1
[zk: zookeeper(CONNECTED) 13] create /config/order-service,zone1/server.port 8090
Created /config/order-service,zone1/server.port
[zk: zookeeper(CONNECTED) 14] create /config/order-service,zone2/server.port 9090
Node does not exist: /config/order-service,zone2/server.port
[zk: zookeeper(CONNECTED) 15] create /config/order-service,zone2 ""
Created /config/order-service,zone2
[zk: zookeeper(CONNECTED) 16] create /config/order-service,zone2/server.port 9090
Created /config/order-service,zone2/server.port
[zk: zookeeper(CONNECTED) 17] ls /config
[order-service,zone1, order-service,zone2]
```

图 10.10　使用给定路径创建 znode 的 Zookeeper 命令列表

## 10.3　小　　结

本章介绍了两个 Spring Cloud 项目的主要功能——Consul 和 Zookeeper。虽然本章的重点不是 Spring Cloud 功能，但也提供了有关如何启动、配置和维护其工具实例的说明。本章讨论了更高级的方案，如使用 Docker 设置由众多成员组成的集群。在这些方案中，开发人员有机会看到 Docker 作为开发工具的真正威力。它允许开发人员只使用 3 个简单命令初始化一个由 3 个成员组成的集群，而无须任何其他配置。

在使用 Spring Cloud 时，Consul 似乎是 Eureka 作为发现服务器的重要替代品，而对于 Zookeeper 则不能作如是观。读者可能已经注意到了，本章对于 Consul 的介绍篇幅要多于 Zookeeper。此外，Spring Cloud 仅将 Zookeeper 视为第二选择，因为与 Spring Cloud Consul 相比，Zookeeper 仍然没有分区机制或观察实现的配置更改功能。Consul 是一种现代解决方案，旨在满足最新架构的需求，如基于微服务的系统；而 Zookeeper 是一种键/值存储，用作在分布式环境中运行的应用程序的服务发现工具。但是，如果在系统中使用 Apache Foundation 堆栈，则值得考虑使用此工具。由于这一点的存在，开发人员可以利用 Zookeeper 与其他 Apache 组件（如 Camel 或 Karaf）之间的集成，轻松发现使用 Spring Cloud 框架创建的服务。

总而言之，在阅读完本章之后，开发人员应该能够在基于微服务的架构中使用 Spring Cloud Consul 和 Spring Cloud Zookeeper 的主要功能。开发人员还应该了解 Spring Cloud 中所有可用发现和配置工具的主要优点和缺点，以便为系统选择最合适的解决方案。

# 第 11 章　消息驱动的微服务

我们已经讨论了 Spring Cloud 提供的基于微服务架构的许多功能。但是，我们一直在考虑的其实都是基于 RESTful 的同步通信服务。在本书第 1 章“微服务简介”中曾经提到过，还有其他一些流行的通信方式，如发布/订阅或异步、事件驱动的点对点消息传递等，后者就是本章将要介绍的一种微服务实现方法，它和前面章节所介绍的基于 RESTful 的同步通信服务有所不同。

本章还将详细讨论如何使用 Spring Cloud Stream 来构建消息驱动的微服务。

本章将要讨论的主题包括：

- 与 Spring Cloud Stream 相关的主要术语和概念。
- 使用 RabbitMQ 和 Apache Kafka 消息代理作为绑定器。
- Spring Cloud Stream 编程模型。
- 绑定、生成器和使用者的高级配置。
- 扩展、分组和分区机制的实现。
- 多个绑定器支持。

## 11.1　了解 Spring Cloud Stream

Spring Cloud Stream 构建于 Spring Boot 之上。它允许开发人员创建独立的、生产级的 Spring 应用程序，并使用 Spring Integration 来帮助实现与消息代理的通信。使用 Spring Cloud Stream 创建的每个应用程序都可以通过输入和输出通道与其他微服务集成。

这些通道通过与中间件相关的绑定器（Binder）实现连接到外部消息代理。有两种内置的绑定器实现——Kafka 和 Rabbit MQ。

Spring Integration 扩展了 Spring 编程模型，以支持众所周知的企业集成模式（Enterprise Integration Patterns，EIP）。EIP 定义了许多通常用于分布式系统中的协作的组件。读者可能已经听说过消息通道、路由器、聚合器或端点等模式。Spring Integration 框架的主要目标是提供一个基于 EIP 构建 Spring 应用程序的简单模型。如果读者对有关 EIP 的更多详细信息感兴趣，请访问网站 http://www.enterpriseintegrationpatterns.com/patterns/messaging/toc.html。

# 11.2　构建消息传递系统

我们认为引入主要 Spring Cloud Stream 功能的最合适方式是通过基于微服务的示例系统。我们将轻松修改前面章节中讨论过的系统架构。这里不妨简要回顾一下这种架构。我们的系统负责处理订单。它由 4 个独立的微服务组成。order-service 微服务首先与 product-service 服务进行通信，以便收集所选产品的详细信息，然后通过 customer-service 服务来检索有关客户及其账户的信息。现在，发送到 order-service 服务的订单将被异步处理。此时仍有一个公开的 RESTful HTTP API 端点用于客户端提交新订单，但应用程序不会处理它们。它只保存新订单，将其发送到消息代理，然后给客户端发送响应，表示订单已被批准处理。当前讨论的示例的主要目标是显示点对点通信，因而消息将仅由一个应用程序（account-service 服务）接收。图 11.1 说明了这个示例系统的架构。

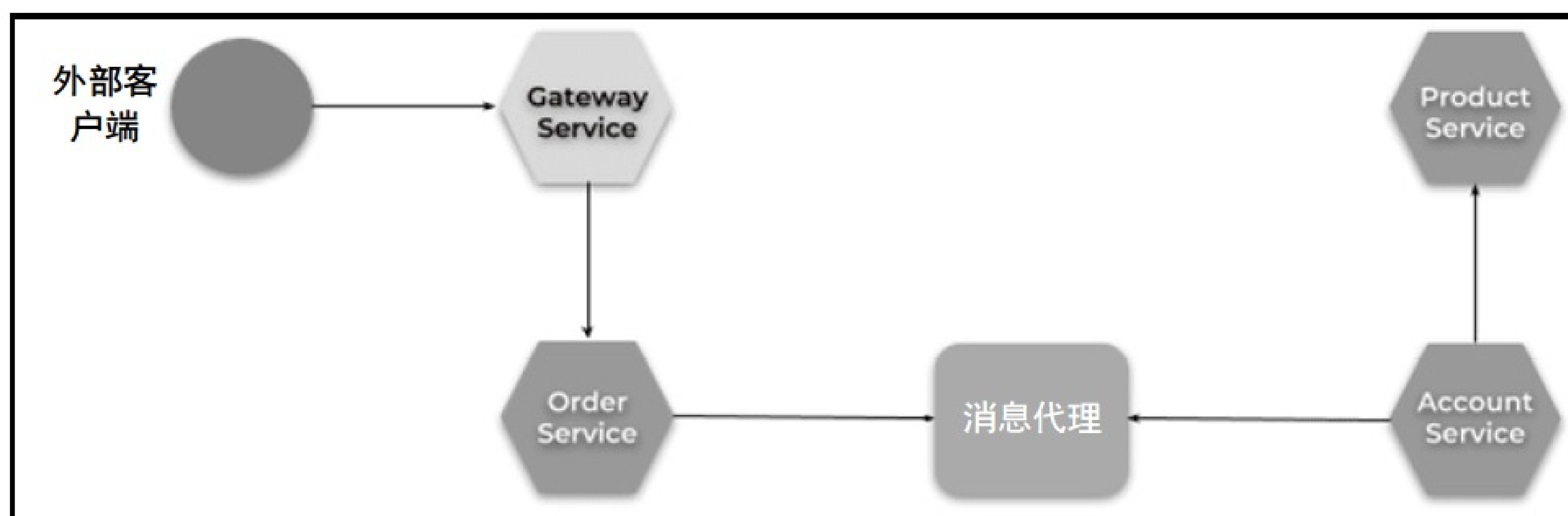



图 11.1　示例系统的架构

收到新消息之后，account-service 服务会调用 product-service 公开的方法以查找其价格。它从账户中提取资金，然后将响应发送回 order-service 服务（包含当前订单的状态）。该消息也通过消息代理发送。order-service 微服务将接收消息并更新订单状态。如果外部客户端想要检查当前订单状态，它可以调用公开 find 方法的端点，查找订单的详细信息。该示例应用程序的源代码可在 GitHub（https://github.com/piomin/sample-spring-cloud-messaging.git）上获得。

## 11.2.1　启用 Spring Cloud Stream

在项目中包含 Spring Cloud Stream 的推荐方法是使用依赖项管理系统。Spring Cloud

Stream 具有与整个 Spring Cloud 框架相关的独立版本列车管理。但是，如果在 dependencyManagement 部分的 Edgware.RELEASE 版本中声明了 spring-cloud-dependencies，那么就不必在 pom.xml 中声明任何其他内容。如果开发人员只想使用 Spring Cloud Stream 项目，则应定义以下部分。

```
<dependencyManagement>
 <dependencies>
  <dependency>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-stream-dependencies</artifactId>
   <version>Ditmars.SR2</version>
   <type>pom</type>
   <scope>import</scope>
  </dependency>
 </dependencies>
</dependencyManagement>
```

下一步是将 spring-cloud-stream 添加到项目依赖项中。此外，建议开发人员至少包含 spring-cloud-sleuth 库，以提供发送消息功能和 traceId，这个 traceId 与通过 Zuul 网关传入 order-service 服务的源请求的 traceId 相同。

```
<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-stream</artifactId>
</dependency>
<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-sleuth</artifactId>
</dependency>
```

要为应用程序启用与消息代理的连接，请使用@EnableBinding 注解主类。@EnableBinding 注解可以将一个或多个接口作为参数。可以在 Spring Cloud Stream 提供的 3 个接口之间选择。

- Sink：用于标记从入站通道接收消息的服务。
- Source：用于向出站频道发送消息。
- Processor：可用于需要入站通道和出站通道的情况，因为它扩展了 Source 和 Sink 接口。由于 order-service 服务发送消息以及接收消息，因而其主类已使用@EnableBinding（Processor.class）进行注解。

以下是支持 Spring Cloud Stream 绑定的主要 order-service 服务类。

```
@SpringBootApplication
@EnableDiscoveryClient
@EnableBinding(Processor.class)
public class OrderApplication {

    public static void main(String[] args) {
        new
SpringApplicationBuilder(OrderApplication.class).web(true).run(args);
    }

}
```

## 11.2.2　声明和绑定频道

由于使用了 Spring Integration，因而该应用程序独立于项目中包含的消息代理实现。Spring Cloud Stream 将自动检测并使用类路径中找到的绑定器，这意味着开发人员可以选择不同类型的中间件，并配合相同的代码使用。所有与中间件相关的设置都可以通过外部配置属性覆盖，并且采用 Spring Boot 支持的形式，如应用程序参数、环境变量或 application.yml 文件。

如前文所述，Spring Cloud Stream 为 Kafka 和 Rabbit MQ 提供了绑定器实现。要包含对 Kafka 的支持，请将以下依赖项添加到项目中。

```
<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-starter-stream-kafka</artifactId>
</dependency>
```

就个人而言，笔者更喜欢 RabbitMQ，本章将为 RabbitMQ 和 Kafka 各创建一个示例。由于前面的章节已经讨论过 RabbitMQ 的功能，所以现在就先从基于 RabbitMQ 的示例开始。

```
<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-starter-stream-rabbit</artifactId>
</dependency>
```

启用 Spring Cloud Stream 并包含绑定器实现后，即可创建发送消息者（Sender）和侦听消息者（Listener）。现在可以从负责向代理发送新订单消息的生产者（Producer）开始。这是通过 order-service 中的 OrderSender 实现的，它使用 Output bean 发送消息。

```
@Service
public class OrderSender {
```

```
    @Autowired
    private Source source;

    public boolean send(Order order) {
        return
this.source.output().send(MessageBuilder.withPayload(order).build());
    }

}
```

该 bean 由控制器调用，它公开允许提交新订单的 HTTP 方法。

```
@RestController
public class OrderController {

    private static final Logger LOGGER =
LoggerFactory.getLogger(OrderController.class);
    private ObjectMapper mapper = new ObjectMapper();

    @Autowired
    OrderRepository repository;
    @Autowired
    OrderSender sender;

    @PostMapping
    public Order process(@RequestBody Order order) throws
JsonProcessingException {
        Order o = repository.add(order);
        LOGGER.info("Order saved: {}", mapper.writeValueAsString(order));
        boolean isSent = sender.send(o);
        LOGGER.info("Order sent: {}",
mapper.writeValueAsString(Collections.singletonMap("isSent", isSent)));
        return o;
    }

}
```

包含订单信息的消息已发送到消息代理。现在，它应该通过 account-service 服务接收。要完成这一操作，必须声明接收器，接收器将侦听传入队列的消息，这个消息是在消息代理上创建的。要接收带有订单数据的消息，只需要使用@StreamListener 注解让该方法采用 Order 对象作为参数。

```
@SpringBootApplication
@EnableDiscoveryClient
@EnableBinding(Processor.class)
public class AccountApplication {

    @Autowired
    AccountService service;

    public static void main(String[] args) {
        new
SpringApplicationBuilder(AccountApplication.class).web(true).run(args);
    }

    @Bean
    @StreamListener(Processor.INPUT)
    public void receiveOrder(Order order) throws JsonProcessingException {
        service.process(order);
    }

}
```

现在可以启动示例应用程序。但是，这里还有一个尚未提及的重要细节。这两个应用程序都尝试连接在 localhost 上运行的 RabbitMQ，并且它们都将相同的交换（Exchange）信息视为输入或输出。这是一个问题，因为 order-service 服务将消息发送到输出交换信息，而 account-service 服务又将侦听传入其输入交换消息。这些是不同的交换信息，但先者恒先。接下来不妨就从运行消息代理开始。

### 11.2.3　自定义与 RabbitMQ 代理的连接

在前面的章节中，已经介绍了使用 Docker 镜像启动 RabbitMQ 代理的方法，因而有必要记住这个命令。它将启动一个带 RabbitMQ 的独立 Docker 容器，可在端口 5672 下使用，其用户界面 Web 控制台可在端口 15672 下使用。

```
docker run -d --name rabbit -p 15672:15672 -p 5672:5672 rabbitmq:management
```

应使用 application.yml 文件中的 spring.rabbit.*属性覆盖默认的 RabbitMQ 地址。

```
spring:
 rabbitmq:
  host: 192.168.99.100
  port: 5672
```

默认情况下，Spring Cloud Stream 会为通信创建主题交换信息。这种类型的交换更适合发布/订阅交互模型。开发人员也可以使用 exchangeType 属性覆盖它，就像在 application.yml 的片段中一样，如下所示。

```
spring:
 cloud:
  stream:
   rabbit:
    bindings:
     output:
      producer:
       exchangeType: direct
     input:
      consumer:
       exchangeType: direct
```

开发人员应该为 order-service 服务和 account-service 服务提供相同的配置设置。这里不必手动创建任何交换信息。如果它不存在，则它会在启动期间由应用程序自动创建。否则，应用程序只会绑定到该交换信息。默认情况下，它创建交换信息时所使用的名称，对于@Input 通道就是输入的名称，对于@Output 通道就是输出的名称。这些名称可以用 spring.cloud.stream.bindings.output.destination 和 spring.cloud.stream.bindings.input.destination 属性覆盖，其中的输入和输出都是通道的名称。此配置选项不仅是 Spring Cloud Stream 功能的一个很好的补充，而且是用于关联服务间通信中的输入和输出目标的键值设置。要解释为什么会发生这种情况也非常简单。在我们的示例中，一方面，order-service 是消息源应用程序，因而它会将消息发送到输出通道。然后，另一方面，account-service 服务将侦听输入通道上的传入消息。如果 order-service 服务的输出通道和 account-service 服务的输入通道未引用代理上的相同目标，则它们之间的通信将失败。总之，我们决定使用名称为 order-out 和 orders-in 的目的地，并且为 order-service 服务提供了以下配置。

```
spring:
 cloud:
  stream:
   bindings:
    output:
     destination: orders-out
    input:
     destination: orders-in
```

account-service 服务的类似配置设置则刚好相反。

```
spring:
 cloud:
  stream:
   bindings:
    output:
     destination: orders-in
    input:
     destination: orders-out
```

在两个应用程序都启动之后，可以使用其 Web 管理控制台轻松查看 RabbitMQ 代理上声明的交换列表，该控制台位于 http://192.168.99.100:15672（guest/guest）。如图 11.2 所示的是隐式创建的交换信息（Exchange），开发人员可能会看到出于测试目的创建的两个目标。

RabbitMQ 3.6.14 Erlang 19.2.1

Overview Connections Channels Exchanges Queues Admin

Exchanges

All exchanges (10)

Pagination

Page 1 of 1 - Filter: Regex ?

| Name | Type | Features | Message rate in | Message rate out | +/- |
|---|---|---|---|---|---|
| (AMQP default) | direct | D | | | |
| amq.direct | direct | D | | | |
| amq.fanout | fanout | D | | | |
| amq.headers | headers | D | | | |
| amq.match | headers | D | | | |
| amq.rabbitmq.log | topic | D I | | | |
| amq.rabbitmq.trace | topic | D I | | | |
| amq.topic | topic | D | | | |
| orders-in | direct | D | | | |
| orders-out | direct | D | | | |

图 11.2　查看 RabbitMQ 代理上声明的交换列表

默认情况下，Spring Cloud Stream 将提供一个输入和一个输出消息通道。可以想象一种情况，即我们的系统需要为每种类型的消息通道提供多个目的地。现在可以回到示例

系统架构，并考虑每个订单由另外两个微服务异步处理的情况。到目前为止，只有 account-service 服务一直在监听来自 order-service 服务的传入事件。在当前示例中，product-service 服务将是传入订单的接收者，它在这种情况下的主要目标是管理可用产品的数量，并根据订单细节减少它们。它要求我们在 order-service 服务中定义两个输入和输出消息通道，因为我们仍然有基于直接 RabbitMQ 交换信息的点对点通信，其中每个消息都可以由一个使用者处理。

在这种情况下，我们应该使用@Input 和@Output 方法声明两个接口。每个方法都必须返回一个 channel 对象。Spring Cloud Stream 提供两个可绑定的消息组件——用于出站通信的 MessageChannel，以及它的扩展——用于入站通信的 SubscribableChannel。这是与 product-service 服务交互的接口定义。已创建用于通过 account-service 服务进行消息传递的类似接口。

```
public interface ProductOrder {

    @Input
    SubscribableChannel productOrdersIn();

    @Output
    MessageChannel productOrdersOut();
}
```

下一步是通过使用@EnableBinding(value={AccountOrder.class,ProductOrder.class})来注解其主类以激活应用程序的声明组件。现在，可以使用它们的名称在配置属性中引用这些通道，如 spring.cloud.stream.bindings.productOrdersOut.destination=product-orders-in。使用@Input 和@Output 注解时，可以通过指定通道名称来自定义每个通道名称，示例如下。

```
public interface ProductOrder {

    @Input("productOrdersIn")
    SubscribableChannel ordersIn();

    @Output("productOrdersOut")
    MessageChannel ordersOut();

}
```

基于自定义接口声明，Spring Cloud Stream 将生成实现该接口的 bean。但是，仍然必须在负责发送消息的 bean 中访问它。与前面的示例相比，直接注入绑定通道会更简便。以下是当前产品订单消息发送者的 bean 实现。还有一个类似的 bean 实现，它可以将消息

发送到 account-service。

```
@Service
public class ProductOrderSender {

    @Autowired
    private MessageChannel output;
    @Autowired
    public SendingBean(@Qualifier("productOrdersOut") MessageChannel
output) {
        this.output = output;
    }

    public boolean send(Order order) {
        return this.output.send(MessageBuilder.withPayload(order).build());
    }

}
```

还应为目标服务提供每个消息通道的自定义接口。侦听消息者应绑定到消息代理上的正确消息通道和目标。

```
@StreamListener(ProductOrder.INPUT)
public void receiveOrder(Order order) throws JsonProcessingException {
    service.process(order);
}
```

## 11.2.4　与其他 Spring Cloud 项目集成

你可能已经注意到，示例系统混合了不同类型的服务间通信。有些微服务使用典型的 RESTful HTTP API，还有一些使用消息代理。在单个应用程序中混合不同类型的通信也没有异议。例如，可以使用 Spring Cloud Stream 将 spring-cloud-starter-feign 包含在项目中，并使用@EnableFeignClients 注解启用它。在我们的示例系统中，这两种不同的通信方式结合了 account-service 服务，它通过消息代理与 order-service 服务集成，并通过 REST API 与 product-service 服务集成。以下是 account-service 服务模块中 Feign 客户端的 product-service 服务实现。

```
@FeignClient(name = "product-service")
public interface ProductClient {
    @PostMapping("/ids")
```

```
    List<Product> findByIds(@RequestBody List<Long> ids);
}
```

还有其他好消息。由于 Spring Cloud Sleuth 的存在，在通过网关传入系统的单个请求期间交换的所有消息都具有相同的 traceId。无论是同步 REST 通信还是异步消息传递，开发人员都可以使用标准日志文件或日志聚合器工具（如 Elastic Stack）轻松跟踪和关联微服务之间的日志。

现在是运行和测试示例系统的好时机。首先，开发人员必须使用 mvn clean install 命令构建整个项目。要使用两个微服务来侦听两个不同交换上的消息来访问代码示例，开发人员应该切换到 advanced 分支（https://github.com/piomin/sample-spring-cloud-messaging/tree/advanced）。应该启动其中可用的所有应用程序——网关、发现和 3 个微服务（account-service 服务、order-service 服务、product-service 服务）。目前讨论的案例假设我们还使用其 Docker 容器启动了 RabbitMQ、Logstash、Elasticsearch 和 Kibana。有关如何使用 Docker 镜像在本地运行 Elastic Stack 的详细说明，请参阅本书第 9 章“分布式日志记录和跟踪”。图 11.3 详细显示了本示例系统的架构。

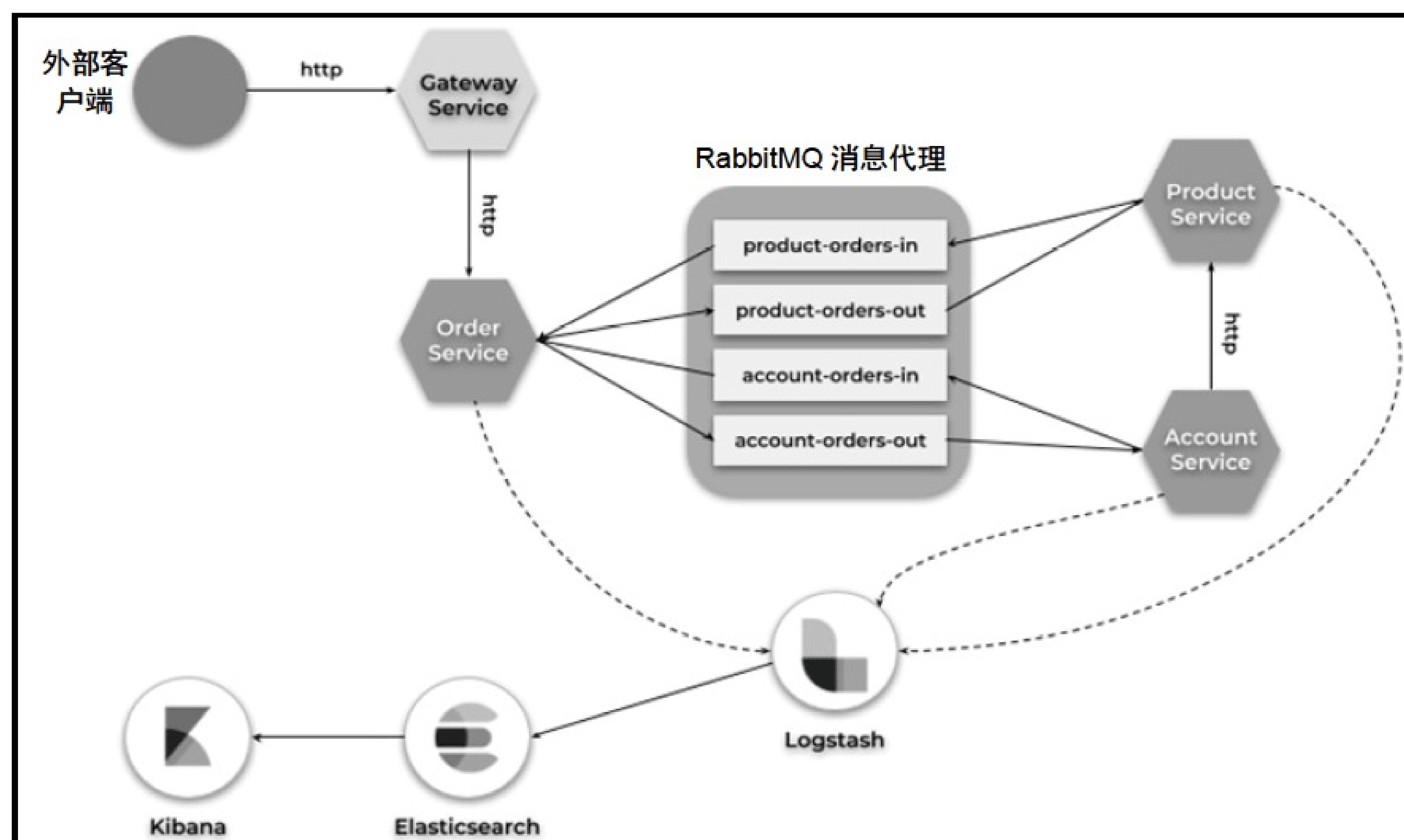



图 11.3　示例系统的架构

运行所有必需的应用程序和工具后，即可继续进行测试。以下是示例请求，它可以通过 API 网关发送到 order-service 服务。

```
curl -H "Content-Type: application/json" -X POST -d
'{"customerId":1,"productIds":[1,3,4],"status":"NEW"}'
http://localhost:8080/api/order
```

如果按照前文所述配置的应用程序运行测试时，第一次可能无法正常工作。有些开发人员可能会有点困惑，因为通常它是在默认设置下测试的。要让它正确运行，还必须在 application.yml 中添加以下属性：spring.cloud.stream.rabbit.bindings.output.producer.routingKeyExpression:'"#"'。它可以将默认生产者的路由键设置为与应用程序引导期间自动创建的交换信息的路由键一致。在如图 11.4 所示的屏幕截图中，可能会看到一个输出交换信息定义。

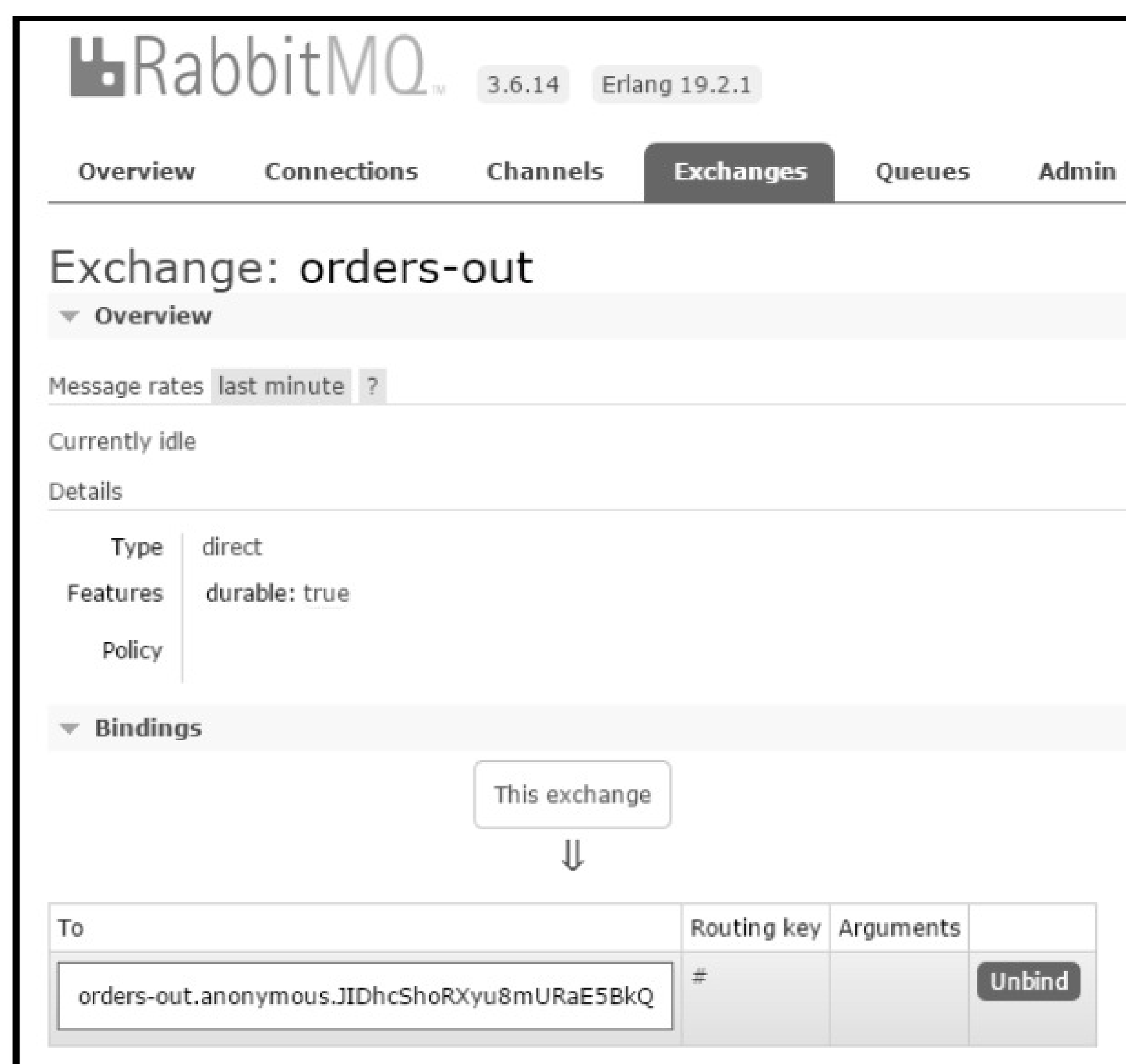



图 11.4　其中的一个输出交换信息

在执行上述修改之后，应该能成功完成测试。微服务打印的日志通过 traceId 相互关联。我们在 logback-spring.xml 中稍微修改了默认的 Sleuth 日志记录格式，这就是现在配置的方式——%d {HH:mm:ss.SSS}%-5level [%X {X-B3-TraceId: - } ,%X {X-B3-SpanId: - }] %msg%n。在发送测试性的 order-service 服务测试请求后，日志将记录以下信息。

```
12:34:48.696 INFO [68038cdd653f7b0b,68038cdd653f7b0b] Order saved:
{"id":1,"status":"NEW","price":0,"customerId":1,"accountId":null,
"productId s":[1,3,4]}
```

```
12:34:49.821 INFO [68038cdd653f7b0b,68038cdd653f7b0b] Order sent:
{"isSent":true}
```

可以看到，account-service 服务也使用相同的日志记录格式，并打印与 order-service 服务相同的 traceId。

```
12:34:50.079 INFO [68038cdd653f7b0b,23432d962ec92f7a] Order processed:
{"id":1,"status":"NEW","price":0,"customerId":1,"accountId":null,
"productId s":[1,3,4]}
12:34:50.332 INFO [68038cdd653f7b0b,23432d962ec92f7a] Account found:
{"id":1,"number":"1234567890","balance":50000,"customerId":1}
12:34:52.344 INFO [68038cdd653f7b0b,23432d962ec92f7a] Products found:
[{"id":1,"name":"Test1","price":1000},{"id":3,"name":"Test3",
"price":2000},{"id":4,"name":"Test4","price":3000}]
```

可以使用 Elastic Stack 聚合在单个事务期间生成的所有日志。可以通过 X-B3-TraceId 字段（如 9da1e5c83094390d）过滤条目，如图 11.5 所示。




图 11.5　聚合日志

# 11.3　发布/订阅模型

事实上，创建 Spring Cloud Stream 项目的主要动机是支持持久的发布/订阅模型。在前面的小节中，我们讨论了微服务之间的点对点通信，这只是一个附加功能。但是，无论我们是否决定使用点对点通信或发布/订阅模型，编程模型仍然是相同的。

在发布/订阅通信中，数据通过共享主题广播。它降低了生产者（Producer）和使用者（Consumer）的复杂性，并允许将新应用程序轻松添加到现有拓扑中，而无须对流程

进行任何更改。这可以在最后提供的系统示例中清楚地看到，在该系统中，我们决定添加第二个应用程序，它将使用源微服务生成的事件。与初始架构相比，开发人员必须定义专用于每个目标应用程序的自定义消息通道。通过队列直接通信，消息只能由一个应用程序实例使用，因此，解决方案是必要的。发布/订阅模型的使用简化了该架构。

### 11.3.1　运行示例系统

要开发采用发布/订阅模型的示例应用程序，比开发采用点对点通信的示例应用程序更简单。开发人员不必覆盖任何默认消息通道以启用与多个接收器的交互。与演示向单个目标应用程序（account-service 服务）传递消息的初始示例相比，这里只需要稍微修改一下配置设置。由于 Spring Cloud Stream 默认绑定到主题，因而不必为输入消息通道覆盖 exchangeType。正如以下配置片段所示，我们仍然在将响应发送到 order-service 服务时使用点对点通信。如果认真思考一下就会发现，这自有其道理。order-service 微服务发送的消息必须由 account-service 服务和 product-service 服务接收，而来自它们的响应仅针对 order-service 服务。

```
spring:
 application:
  name: product-service
 rabbitmq:
  host: 192.168.99.100
  port: 5672
 cloud:
  stream:
   bindings:
  output:
   destination: orders-in
  input:
   destination: orders-out
 rabbit:
  bindings:
   output:
    producer:
     exchangeType: direct
     routingKeyExpression: '"#"'
```

product-service 服务的主要处理方法的逻辑非常简单。它只需要从收到的订单中找到所有的 productId，更改每个产品的库存数量，然后将响应发送到 order-service 服务。

```
@Autowired
ProductRepository productRepository;
@Autowired
OrderSender orderSender;

public void process(final Order order) throws JsonProcessingException {
 LOGGER.info("Order processed: {}", mapper.writeValueAsString(order));
 for (Long productId : order.getProductIds()) {
     Product product = productRepository.findById(productId);
     if (product.getCount() == 0) {
         order.setStatus(OrderStatus.REJECTED);
         break;
     }
     product.setCount(product.getCount() - 1);
     productRepository.update(product);
     LOGGER.info("Product updated: {}",
mapper.writeValueAsString(product));
 }
 if (order.getStatus() != OrderStatus.REJECTED) {
     order.setStatus(OrderStatus.ACCEPTED);
 }
 LOGGER.info("Order response sent: {}",
mapper.writeValueAsString(Collections.singletonMap("status",
order.getStatus())));
 orderSender.send(order);
}
```

要访问当前示例，只需切换到 publish_subscribe 分支，这可从 https://github.com/piomin/sample-spring-cloud-messaging/tree/publish_subscribe 获取。然后，开发人员应该构建父项目并运行与上一个示例相同的所有服务。如果想要让测试一切正常，直到只有一个正在运行的 account-service 服务和 product-service 服务实例，那么现在就可以来讨论这个问题。

## 11.3.2　扩展和分组

在谈论基于微服务的架构时，可伸缩性（Scalability）始终是其主要优势之一。通过创建给定应用程序的多个实例来扩展系统的能力非常重要。执行此操作时，应用程序的不同实例将放置在竞争的使用者关系中，其中只有一个实例需要处理给定的消息。对于点对点通信来说，这不是问题，但在发布-订阅模型中，消息会被所有接收者使用，这可能是一个挑战。

### 1．运行多个实例

扩展微服务实例数量的可用性是 Spring Cloud Stream 的主要概念之一。然而，这个想法背后没有神奇的地方。使用 Spring Cloud Stream 可以非常轻松地运行应用程序的多个实例。其中一个原因是来自消息代理的原生支持，它旨在处理许多使用者和大量流量。

在这种情形下，所有消息传递微服务也将公开 RESTful HTTP API，因此，首先必须为每个实例定制服务器端口。我们之前已经进行了此类操作。还可以考虑设置两个 Spring Cloud Stream 属性 spring.cloud.stream.instanceCount 和 spring.cloud.stream.instanceIndex。多亏了它们，微服务的每个实例都能够接收有关同一应用程序的其他几个示例的启动信息以及它自己的实例索引。仅当要启用分区功能时，才需要正确配置这些属性。下文很快将谈论这个机制。现在，让我们来看一看扩展应用程序的配置设置。account-service 服务和 product-service 服务都定义了两个配置文件，用于运行应用程序的多个实例。我们已经自定义了服务器的 HTTP 端口、实例的数量和索引。

```
---
spring:
 profiles: instance1
 cloud:
  stream:
   instanceCount: 2
   instanceIndex: 0
server:
 port: ${PORT:8091}

---
spring:
 profiles: instance2
 cloud:
  stream:
   instanceCount: 2
   instanceIndex: 1
server:
 port: ${PORT:9091}
```

构建父项目后，开发人员可以运行该应用程序的两个实例。它们中的每一个都使用分配给在启动期间传递的正确配置文件的属性进行初始化，如 java -jar --spring.profiles.active=instance1 target/account-service-1.0-SNAPSHOT.jar。如果向 order-service 服务端点 POST / 发送测试请求，则新订单将转发到 RabbitMQ 主题交换信息，以便由连接到该交换的 account-service 服务和 product-service 服务接收。现在的问题是每个服务的所有实例

都收到消息，这并不是我们想要实现的。要解决这个问题，分组机制可以带来帮助。

### 2．使用者分组

我们的目的很明确。现在有许多微服务使用来自同一主题的消息。应用程序的不同实例被置于竞争的使用者关系中，但只有其中一个应该处理给定的消息。Spring Cloud Stream 引入了模拟此行为的使用者分组（Consumer Group）的概念。要激活此类行为，我们应该使用组名设置为 spring.cloud.stream.bindings.<channelName>.group 的属性。设置之后，订阅给定目标的所有分组都会接收到已发布数据的副本，但每个组中只有一个成员接收并处理来自该目标的消息。在我们的示例中，有两个分组。第一个是具有名称账户的所有 account-service 服务实例的分组；第二个则是具有名称产品的 product-service 服务实例的分组。

以下是 account-service 服务的当前绑定配置。orders-in 目的地是为与 order-service 服务直接通信而创建的队列，因而只有 orders-out 按服务名称分组。为 product-service 服务也准备了类似的配置。

```
spring:
 cloud:
  stream:
   bindings:
    output:
     destination: orders-in
    input:
     destination: orders-out
     group: account
```

第一个区别在为 RabbitMQ 交换信息自动创建的队列名称中可见。现在，它不是随机生成的名称，如 orders-in.anonymous.qNxjzDq5Qra-yqHLUv50PQ，而是由目标和分组名称组成的确定字符串。如图 11.6 所示的屏幕截图显示了 RabbitMQ 上当前存在的所有队列。

| Overview | | | Messages | | | Message rates | | |
|---|---|---|---|---|---|---|---|---|
| Name | Features | State | Ready | Unacked | Total | incoming | deliver / get | ack |
| orders-in.anonymous.qNxjzDq5Qra-yqHLUv50PQ | AD Excl | idle | 24 | 1 | 25 | 2.6/s | 0.20/s | 0.00/s |
| orders-out.account | D | running | 11 | 2 | 13 | 0.20/s | 2.4/s | 2.4/s |
| orders-out.product | D | idle | 0 | 0 | 0 | 0.20/s | 0.20/s | 0.20/s |

图 11.6　在 RabbitMQ 上当前存在的所有队列

开发人员可以自己执行重新测试，以验证该消息是否仅由同一组中的一个应用程序接收。但是，开发人员无法确定哪个实例将处理传入的消息。为了确定这一点，可以考

虑使用分区机制。

### 3．分区机制

Spring Cloud Stream 支持在多个应用程序实例之间对数据进行分区（Partitioning）。在典型的用例中，目标可被划分为不同的分区。每个生产者在发送由多个使用者实例接收的消息时，将确保由配置的字段标识数据以强制由同一使用者实例处理。

要为应用程序启用分区功能，必须在生产者配置设置中定义 partitionKeyExpression 或 partitionKeyExtractorClass 属性以及 partitionCount。以下是可能为应用程序提供的示例配置。

```
spring.cloud.stream.bindings.output.producer.partitionKeyExpression =
payload.customerId
spring.cloud.stream.bindings.output.producer.partitionCount = 2
```

分区机制还需要在使用者端设置 spring.cloud.stream.instanceCount 和 spring.cloud.stream.instanceIndex 属性。还必须将 spring.cloud.stream.bindings.input.consumer.partitioned 属性设置为 true 才能显式启用它。实例索引负责标识特定实例从中接收数据的唯一分区。一般来说，生产者端的 partitionCount 和使用者端的 instanceCount 应该相等。

现在来了解一下由 Spring Cloud Stream 提供的分区机制。首先，它将根据 partitionKeyExpression 计算分区键，该分区键是根据出站消息或 PartitionKeyExtractorStrategy 接口的实现来计算的，该接口定义了用于提取消息的键的算法。计算完消息的键之后，目标分区将被确定为 0 和 partitionCount-1 之间的值。默认计算公式为 key.hashCode()%partitionCount。它可以使用 partitionSelectorExpression 属性进行自定义，也可以创建 org.springframework.cloud.stream.binder.PartitionSelectorStrategy 接口的实现。计算出的键将与使用者端的 instanceIndex 匹配。

在解释了围绕分区机制的主要概念之后，现在可以来看一看其示例。以下是 product-service 服务输入通道的当前配置（与为 account-service 服务设置的账户分组名称相同）。

```
spring:
 cloud:
  stream:
   bindings:
    input:
     consumer:
      partitioned: true
     destination: orders-out
     group: product
```

现在每个微服务都有两个正在运行的实例，它们使用来自主题交换信息的数据。在 order-service 服务中还为生产者设置了两个分区。消息键是根据 Order 对象中的 customerId 字段计算的。索引为 0 的分区专用于 customerId 字段中具有偶数的订单，而索引为 1 的分区则用于 customerId 字段中的奇数订单。

实际上，RabbitMQ 没有对分区的原生支持。有趣的是，Spring Cloud Stream 使用 RabbitMQ 实现分区处理的方式。在如图 11.7 所示的屏幕截图中，显示了在 RabbitMQ 中创建的交换信息的绑定列表。在该图中可见已经为 exchange-orders-out-0 和 orders-out-1 定义了两个路由键。

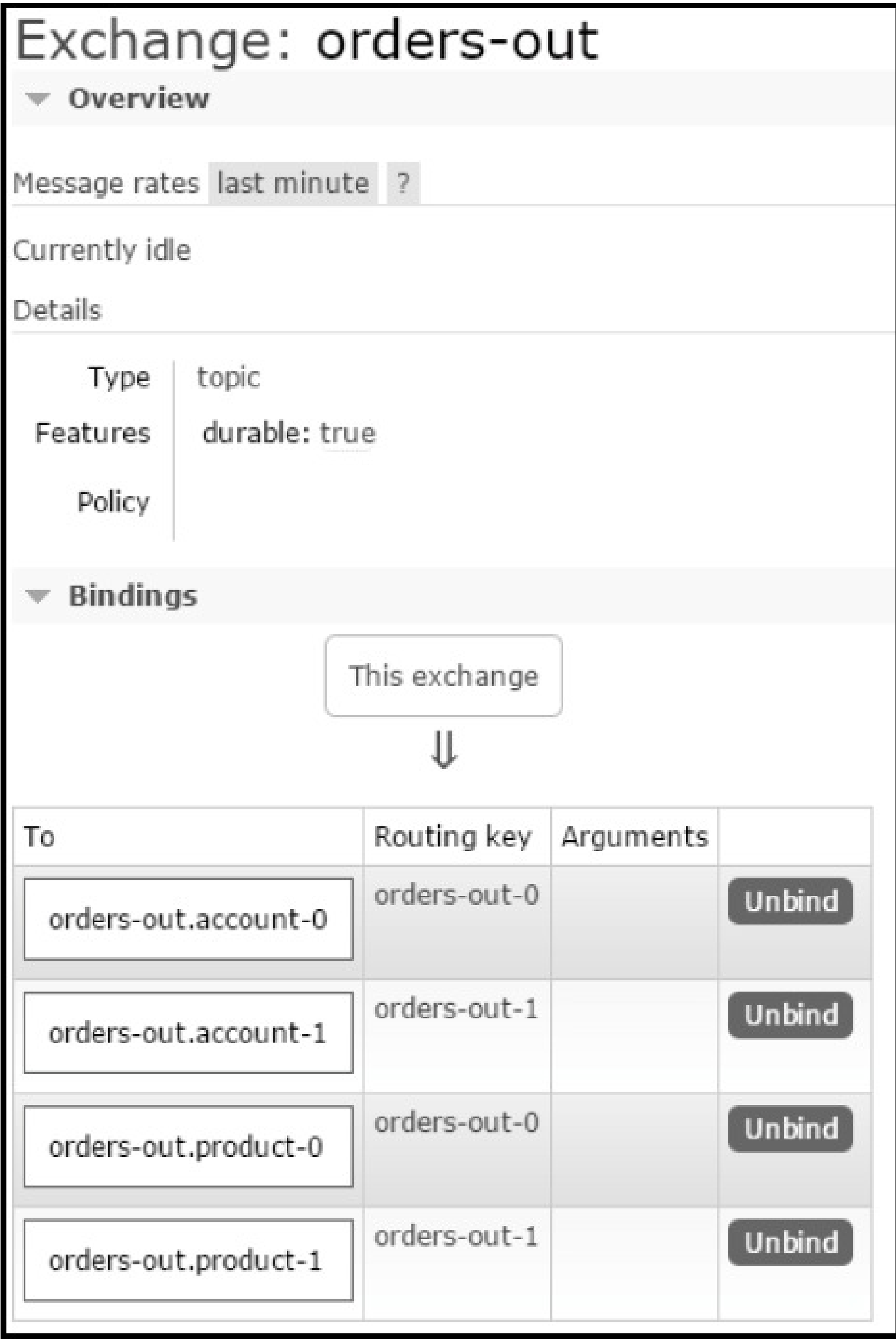



图 11.7　在 RabbitMQ 中创建的交换信息的绑定列表

如果在 JSON 消息中发送了一个 customerId 等于 1 的订单，如{"customerId":1, "productIds":[4],"status":"NEW"}，那么它将始终由 instanceIndex = 1 的实例处理。可以在应用程序日志中或使用 RabbitMQ Web 控制台来查看它。如图 11.8 所示就是一个包含每个队列的消息速率的屏幕截图，可以看到 customerId = 1 的消息已被多次发送。

| Overview | | | Messages | | | Message rates | | |
|---|---|---|---|---|---|---|---|---|
| Name | Features | State | Ready | Unacked | Total | incoming | deliver / get | ack |
| orders-in.anonymous.IoLFDyEMTZCsMl2R7-ac9Q | AD Excl | running | 44 | 1 | 45 | 8.8/s | 0.40/s | 0.00/s |
| orders-out.account-0 | D | idle | 0 | 0 | 0 | | | |
| orders-out.account-1 | D | running | 0 | 0 | 0 | 4.4/s | 4.4/s | 4.4/s |
| orders-out.product-0 | D | idle | 0 | 0 | 0 | | | |
| orders-out.product-1 | D | running | 0 | 0 | 0 | 4.4/s | 4.4/s | 4.4/s |

图 11.8　在 RabbitMQ Web 控制台中查看被处理的消息

# 11.4　配 置 选 项

可以使用 Spring Boot 支持的任何机制（如应用程序参数、环境变量和 YAML 或属性文件）覆盖 Spring Cloud Stream 配置设置。它定义了许多可应用于所有绑定器的通用配置选项。但是，还有一些与应用程序使用的特定消息代理相关的其他属性。

## 11.4.1　Spring Cloud Stream 属性

当前的属性组适用于整个 Spring Cloud Stream 应用程序。表 11.1 中的所有属性都以 spring.cloud.stream 为前缀。

表 11.1　Spring Cloud Stream 属性

| 名　称 | 默 认 值 | 说　明 |
|---|---|---|
| instanceCount | 1 | 应用程序的运行实例数。有关更多详细信息，请参阅第 11.3.2 节“扩展和分组” |
| instanceIndex | 0 | 应用程序实例的索引。有关更多详细信息，请参阅第 11.3.2 节“扩展和分组” |
| dynamicDestinations | - | 可以动态绑定的目标列表 |
| defaultBinder | - | 如果定义了多个绑定器，则为默认绑定器。有关更多详细信息，请参阅第 11.7 节“多个绑定器” |
| overrideCloudConnectors | false | 仅在云处于活动状态且在类路径中找到 Spring Cloud Connectors 时才使用此选项。当它设置为 true 时，绑定器将完全忽略绑定的服务并依赖 spring.rabbitmq.* 或 spring.kafka.* Spring Boot 属性 |

## 11.4.2　绑定属性

下一组属性与消息通道相关。在 Spring Cloud 术语中，这些都是绑定属性。它们可以仅分配给使用者、生产者或同时分配给两者。表 11.2 是绑定属性列表及其默认值和说明。

表 11.2　绑定属性列表及其默认值和说明

| 名　称 | 默 认 值 | 说　明 |
| --- | --- | --- |
| destination | - | 为消息通道配置的代理上的目标名称。如果通道仅由一个使用者使用，则可以将其指定为逗号分隔的目标列表 |
| group | null | 通道的使用者分组。有关更多详情请参阅第 11.3.2 节“扩展和分组” |
| contentType | null | 通过给定通道交换的消息的内容类型。例如，可以将它设置为 application/json。然后，从该应用程序发送的所有对象将自动转换为 JSON 字符串 |
| binder | null | 通道使用的默认绑定器。有关更多详情请参阅第 11.7 节“多个绑定器” |

### 1．使用者

以下属性列表仅适用于输入绑定，并且必须以 spring.cloud.stream.bindings.<channelName>.consumer 为前缀。其中最重要的一些属性如表 11.3 所示。

表 11.3　适用于使用者的重要属性

| 名　称 | 默 认 值 | 说　明 |
| --- | --- | --- |
| concurrency | 1 | 每个输入通道的使用者数量 |
| partitioned | false | 它允许接收来自分区的生产者的数据 |
| headerMode | embeddedHeaders | 如果将其设置为 raw，则禁用输入上的标头解析 |
| maxAttempts | 3 | 在消息处理失败之后的重试次数。将此选项设置为 1 将禁用重试机制 |

### 2．生产者

以下绑定属性仅可用于输出绑定，并且必须以 spring.cloud.stream.bindings.<channelName>.producer 为前缀。其中最重要的一些如表 11.4 所示。

表 11.4　适用于生产者的重要属性

| 名　称 | 默 认 值 | 说　明 |
| --- | --- | --- |
| requiredGroups | - | 必须在消息代理上创建的以逗号分隔的分组列表 |
| headerMode | embeddedHeaders | 如果将其设置为 raw，则禁用输入上的标头解析 |
| useNativeEncoding | false | 如果将其设置为 true，则出站消息将由客户端库直接序列化 |
| errorChannelEnabled | false | 如果将其设置为 true，则将失败消息发送到目标的错误通道 |

# 11.5　高级编程模型

Spring Cloud Stream 编程模型的基础知识与点对点和发布/订阅通信的示例将一起介绍。现在来讨论一些更高级的示例功能。

## 11.5.1　制作消息

在本章介绍的所有示例中，我们已通过 RESTful API 发送订单以进行测试。但是，开发人员也可以通过在应用程序内定义消息源来轻松创建一些测试数据。以下是一个使用@Poller 的 bean，它将每秒生成一条消息，并将其发送到输出通道。

```
@Bean
@InboundChannelAdapter(value = Source.OUTPUT, poller = @Poller(fixedDelay =
"1000", maxMessagesPerPoll = "1"))
public MessageSource<Order> ordersSource() {
    Random r = new Random();
    return () -> new GenericMessage<>(new Order(OrderStatus.NEW, (long)
r.nextInt(5), Collections.singletonList((long) r.nextInt(10))));
}
```

## 11.5.2　转换

如前文所述，account-service 服务和 product-service 服务已经从 order-service 服务接收事件，然后发回响应消息。我们创建了 OrderSender bean，它负责准备响应有效负载并将其发送到输出通道。事实证明，如果在方法中返回响应对象并使用@SentTo 注解它，则实现可能会更简单。

```
@StreamListener(Processor.INPUT)
@SendTo(Processor.OUTPUT)
public Order receiveAndSendOrder(Order order) throws
JsonProcessingException {
    LOGGER.info("Order received: {}", mapper.writeValueAsString(order));
    return service.process(order);
}
```

我们甚至可以想象诸如以下形式的实现，而不使用@StreamListener。转换器（Transformer）模式将负责更改对象的形式。在这种情况下，它会修改两个 order 字段——status（状态）

和 price（价格）。

```
@EnableBinding(Processor.class)
public class OrderProcessor {

    @Transformer(inputChannel = Processor.INPUT, outputChannel =
Processor.OUTPUT)
    public Order process(final Order order) throws JsonProcessingException{
        LOGGER.info("Order processed: {}",mapper.writeValueAsString(order));
        // ...
        products.forEach(p -> order.setPrice(order.getPrice() +
p.getPrice()));
        if (order.getPrice() <= account.getBalance()) {
            order.setStatus(OrderStatus.ACCEPTED);
            account.setBalance(account.getBalance() - order.getPrice());
        } else {
            order.setStatus(OrderStatus.REJECTED);
        }
        return order;
    }

}
```

## 11.5.3　有条件地使用消息

假设开发人员希望以不同方式处理传入同一消息通道的消息，则可以使用条件分派。Spring Cloud Stream 支持根据条件将消息分派给在输入通道上注册的多个@StreamListener 方法。该条件是在@StreamListener 注解的 condition 属性中定义的 Spring 表达式语言（Spring Expression Language，SpEL）表达式。

```
public boolean send(Order order) {
    Message<Order> orderMessage =
MessageBuilder.withPayload(order).build();
    orderMessage.getHeaders().put("processor", "account");
    return this.source.output().send(orderMessage);
}
```

以下就是一个示例实现，它定义了两个使用@StreamListener 注解的方法，这些方法侦听同一主题。其中一个专用于从 account-service 服务传入的消息，而第二个则专用于 product-service 服务。传入消息将根据带有 processor 名称的标头进行分派。

```
@SpringBootApplication
@EnableDiscoveryClient
@EnableBinding(Processor.class)
public class OrderApplication {

    @StreamListener(target = Processor.INPUT, condition =
"headers['processor']=='account'")
    public void receiveOrder(Order order) throws JsonProcessingException {
        LOGGER.info("Order received from account: {}",
mapper.writeValueAsString(order));
        // ...
    }

    @StreamListener(target = Processor.INPUT, condition =
"headers['processor']=='product'")
    public void receiveOrder(Order order) throws JsonProcessingException {
        LOGGER.info("Order received from product: {}",
mapper.writeValueAsString(order));
        // ...
    }

}
```

## 11.6　使用 Apache Kafka

在讨论 Spring Cloud 与消息代理的集成时，我们曾经多次提到过 Apache Kafka。但是，到目前为止，我们还没有基于该平台运行任何示例。事实上，RabbitMQ 在使用 Spring Cloud 项目时往往是首选，但是 Kafka 也值得我们关注。与 RabbitMQ 相比，它的一个优势是对分区的原生支持，而分区正是 Spring Cloud Stream 最重要的功能之一。

Kafka 不是典型的消息代理。它是一个分布式流媒体平台。它的主要功能是允许开发人员发布和订阅记录（Record）流。它对转换或响应数据流的实时流应用程序特别有用。它通常作为由一个或多个服务器组成的集群运行，并可以在主题中存储记录流。

### 11.6.1　运行 Kafka

糟糕的是，Apache Kafka 没有正式的 Docker 镜像。但是，我们可以使用一个非官方的，如 Spotify 共享的镜像。与其他可用的 Kafka docker 镜像相比，这个镜像可以在同一

容器中运行 Zookeeper 和 Kafka。以下是启动 Kafka 并在端口 9092 上公开它的 Docker 命令。在端口 2181 上也可以使用 Zookeeper。

```
docker run -d --name kafka -p 2181:2181 -p 9092:9092 --env
ADVERTISED_HOST=192.168.99.100 --env ADVERTISED_PORT=9092 spotify/kafka
```

## 11.6.2　自定义应用程序设置

要为应用程序启用 Apache Kafka，需要将 spring-cloud-starter-stream-kafka 启动程序包含在依赖项中。我们当前的示例非常类似于发布/订阅的示例，因为它使用了 RabbitMQ 发布/订阅，以及在第 11.3 节“发布/订阅模型”中介绍过的分组和分区机制。唯一的区别在于依赖项和配置设置。

Spring Cloud Stream 将自动检测并使用类路径中找到的绑定器。可以使用 spring.kafka.*属性覆盖连接设置。在这种情况下中，只需要将自动配置的 Kafka 客户端地址更改为 Docker 机器地址 192.168.99.100。对 Zookeeper 也应该执行相同的修改，因为 Zookeeper 将由 Kafka 客户端使用。

```
spring:
 application:
  name: order-service
  kafka:
   bootstrap-servers: 192.168.99.100:9092
 cloud:
  stream:
   bindings:
    output:
     destination: orders-out
     producer:
      partitionKeyExpression: payload.customerId
      partitionCount: 2
    input:
     destination: orders-in
   kafka:
    binder:
     zkNodes: 192.168.99.100
```

在启动发现、网关和所有必需的微服务实例后，即可执行与先前示例相同的测试。如果一切配置正确，则应该在应用程序启动期间在日志中看到以下片段。其测试结果与基于 RabbitMQ 的示例完全相同。

```
16:58:30.008 INFO [,] Discovered coordinator 192.168.99.100:9092
(id: 2147483647 rack: null) for group account.
16:58:30.038 INFO [,] Successfully joined group account with generation 1
16:58:30.039 INFO [,] Setting newly assigned partitions
[orders-out-0, orders-out-1] for group account
16:58:30.081 INFO [,] partitions assigned:
[orders-out-0, orders-out-1]
```

### 11.6.3　Kafka Streams API 支持

Spring Cloud Stream Kafka 可以提供专为 Kafka Streams 绑定设计的绑定器。使用此绑定器之后，应用程序即可利用 Kafka Streams API。要为应用程序启用此类功能，需要在项目中包含以下依赖项。

```
<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-stream-binder-kstream</artifactId>
</dependency>
```

Kafka Streams API 可以提供高级流 DSL。可以通过声明@StreamListener 方法将 KStream 接口作为参数来访问它。KStream 为流操作提供了一些有用的方法，这些方法来自其他流 API，如 map、flatMap、join 或 filter。还有一些与 Kafka Stream 相关的其他方法，如 to(...)（用于向主题发送流）或 through(...)（与 to 相同，但也会从主题创建 KStream 的新实例）。

```
@SpringBootApplication
@EnableBinding(KStreamProcessor.class)
public class AccountApplication {

    @StreamListener("input")
    @SendTo("output")
    public KStream<?, Order> process(KStream<?, Order> input) {
        // ..
    }

    public static void main(String[] args) {
        SpringApplication.run(AccountApplication.class, args);
    }

}
```

### 11.6.4　配置属性

在讨论示例应用程序的实现之前，我们已经介绍了 Kafka 的一些 Spring Cloud 配置设置。表 11.5 包含了一些最重要的属性，可以设置这些属性来自定义 Apache Kafka 绑定器。所有这些属性都以 spring.cloud.stream.kafka.binder 为前缀。

表 11.5　Kafka 的重要配置属性

| 名　称 | 默 认 值 | 说　明 |
| --- | --- | --- |
| brokers | localhost | 以逗号分隔的代理列表，包含或不包含端口信息 |
| defaultBrokerPort | 9092 | 如果没有使用 brokers 属性定义端口，则设置默认端口 |
| zkNodes | localhost | 以逗号分隔的 ZooKeeper 节点列表，包含或不包含端口信息 |
| defaultZkPort | 2181 | 如果没有使用 zkNodes 属性定义端口，则设置默认的 ZooKeeper 端口 |
| configuration | - | Kafka 客户端属性的键/值映射。它适用于由绑定器创建的所有客户端 |
| headers | - | 将由绑定器转发的自定义标头列表 |
| autoCreateTopics | true | 如果设置为 true，则绑定器会自动创建新主题 |
| autoAddPartitions | false | 如果设置为 true，则绑定器会自动创建新分区 |

## 11.7　多个绑定器

在 Spring Cloud Stream 术语中，可以实现以提供与外部中间件的物理目标的连接的接口称为绑定器（Binder）。目前，有两种可用的内置绑定器实现——Kafka 和 RabbitMQ。如果想要提供自定义绑定器库，那么关键接口就是 Binder（这个关键接口其实就是作为将输入和输出连接到外部中间件的策略的抽象），它有两个方法——bindConsumer 和 bindProducer。有关更多详细信息，请参阅 Spring Cloud Stream 规范。

对开发人员来说，重要的是能够在单个应用程序中使用多个绑定器。我们甚至可以混合使用不同的实现，如 RabbitMQ 和 Kafka。Spring Cloud Stream 依赖于 Spring Boot 在绑定过程中的自动配置。类路径上可用的实现将自动使用。如果想要同时使用默认的绑定器，请在项目中包含以下依赖项。

```
<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-stream-binder-rabbit</artifactId>
```

```
</dependency>
<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-stream-binder-kafka</artifactId>
</dependency>
```

如果在类路径中找到了多个绑定器，则应用程序必须检测应将哪个绑定器用于特定通道绑定。我们可以使用 spring.cloud.stream.defaultBinder 属性全局配置默认绑定器，或者使用 spring.cloud.stream.bindings.<channelName>.binder 属性为每个通道单独配置默认绑定器。现在不妨回到之前的示例，在那里配置多个绑定器。我们需要为 account-service 服务和 order-service 服务之间的直接通信定义 RabbitMQ，并为 order-service 服务和其他微服务之间的发布/订阅模型定义 Kafka。

以下是与 publish_subscribe 分支（https://github.com/piomin/sample-spring-cloud-messaging/tree/publish_subscribe）中的 account-service 相同的配置，但它基于两个不同的绑定器。

```
spring:
 cloud:
  stream:
   bindings:
    output:
     destination: orders-in
     binder: rabbit1
    input:
     consumer:
      partitioned: true
     destination: orders-out
     binder: kafka1
     group: account
   rabbit:
    bindings:
     output:
      producer:
       exchangeType: direct
       routingKeyExpression: '"#"'
   binders:
    rabbit1:
     type: rabbit
     environment:
      spring:
       rabbitmq:
```

```
    host: 192.168.99.100
  kafka1:
   type: kafka
   environment:
    spring:
     kafka:
      bootstrap-servers: 192.168.99.100:9092
```

## 11.8　小　　结

与所有其他 Spring Cloud 项目相比，Spring Cloud Stream 可以被视为一个单独的类别。它通常与其他项目相关联，并且目前由 Pivotal Spring Cloud Data Flow 强力推广。这是用于构建数据集成和实时数据处理管道的工具包。当然，这也是一个巨大的主题，不是单独的一本书就可以讨论完的。

更重要的是，Spring Cloud Stream 支持异步消息传递，并且可以使用 Spring 注解样式轻松实现。我们认为，对于部分开发人员来说，这种服务间通信方式并不像 RESTful API 模型那么明显。因此，我们更专注于展示使用 Spring Cloud Stream 进行点对点和发布/订阅通信的示例。我们还详细介绍了这两种消息传递方式之间的差异。

发布/订阅模型并不是什么新鲜事物，但是由于 Spring Cloud Stream 的存在，它可以很容易地包含在基于微服务的系统中。本章还介绍了一些关键概念，如使用者分组或分区。阅读完本章之后，开发人员应该能够基于消息传递模型实现微服务，并将它们与其他 Spring Cloud 库集成，以便提供日志记录、跟踪或仅将它们部署为现有的基于 REST 的微服务系统的一部分。

# 第 12 章　保护 API 的安全

安全性是与基于微服务的架构相关的最常讨论的问题之一。对于所有与安全相关的话题来说，始终绕不开一个主要问题，那就是网络。对于微服务，通常网络上的通信比一体化应用程序要多得多，所以应该重新考虑认证和授权的方法。传统系统通常采用构筑边界的形式进行保护，然后允许前端服务完全访问后端组件。迁移到微服务则迫使开发人员将这种方法改为委托访问管理。

Spring Framework 如何解决基于微服务架构的安全问题？它提供了若干个实现有关身份验证和授权的不同模式的项目。第一个项目是 Spring Security，它是安全的基于 Spring 的 Java 应用程序的事实标准。它由一些子模块组成，可以帮助开发人员开始使用 SAML、OAuth2 或 Kerberos。另外还有 Spring Cloud Security 项目，它提供了若干个组件，允许开发人员将基本的 Spring Security 功能与微服务架构的主要元素集成在一起，如网关、负载均衡器和 REST HTTP 客户端。

本章将介绍保护基于微服务的系统的所有主要组件。我们将从 Eureka 的服务发现开始，然后转到 Spring Cloud Config Server 和服务间通信，最后再讨论 API 网关的安全性。

本章将要讨论的主题包括：

- 为单个 Spring Boot 应用程序配置安全连接。
- 为基于微服务架构的最重要元素启用 HTTPS 通信。
- 加密和解密存储在 Config Server 上的配置文件中的属性值。
- 使用 OAuth2 为微服务进行基于内存的简单身份验证。
- 使用 JDBC 后端存储和 JWT 令牌进行更高级的 OAuth2 配置。
- 在与 Feign 客户端的服务间通信中使用 OAuth2 授权。

接下来就让我们先从基础开始，演示如何创建第一个安全微服务，并且通过 HTTPS 公开其 API。

## 12.1　为 Spring Boot 启用 HTTPS

如果要使用 SSL 并通过 HTTPS 提供 RESTful API，则需要生成证书。实现这一目标的最快方法是通过自签名证书（Self-Signed Certificate），这对于开发模式来说已经足够了。

Java 运行时环境（Java Runtime Environment，JRE）提供了一个简单的证书管理工具——keytool。它位于 JRE_HOME\bin 目录下。以下代码中的命令将生成自签名证书并将其放入 PKCS12 KeyStore。除了 KeyStore 的类型，开发人员还必须设置其有效性、别名和文件名。在开始生成过程之前，keytool 还会询问密码和一些其他信息，如下所示。

```
keytool -genkeypair -alias account-key -keyalg RSA -keysize 2048 -storetype
PKCS12 -keystore account-key.p12 -validity 3650

Enter keystore password:
Re-enter new password:
What is your first and last name?
 [Unknown]: localhost
What is the name of your organizational unit?
 [Unknown]: =
What is the name of your organization?
 [Unknown]: piomin
What is the name of your City or Locality?
 [Unknown]: Warsaw
What is the name of your State or Province?
 [Unknown]: mazowieckie
What is the two-letter country code for this unit?
 [Unknown]: PL
Is CN=localhost, OU=Unknown, O=piomin, L=Warsaw, ST=mazowieckie, C=PL
correct?
 [no]: yes
```

我们已将生成的证书复制到 Spring Boot 应用程序内的 src/main/resources 目录中。构建并运行应用程序后，它将在类路径中可用。要启用 SSL，必须在 application.yml 文件中提供一些配置设置。通过设置各种 server.ssl.*属性，可以为 Spring 自定义 SSL。

```
server:
 port: ${PORT:8090}

ssl:
 key-store: classpath:account-key.p12
 key-store-password: 123456
 key-store-type: PKCS12
 key-alias: account-key

security:
 require-ssl: true
```

# 12.2　保证发现服务器的安全

如前文所述，微服务应用程序的 SSL 配置并不是一项非常艰巨的任务。但是，是时候提高其难度等级了。我们已经启动一个通过 HTTPS 提供 RESTful API 的单一微服务。现在希望微服务与发现服务器集成。由此产生了两个问题。第一个是需要在 Eureka 上发布有关安全微服务实例的信息；第二个问题涉及通过 HTTPS 暴露 Eureka 并强制发现客户端使用私钥对发现服务器进行身份验证。接下来我们将详细讨论这些问题。

## 12.2.1　注册安全的应用程序

如果应用程序通过安全的 SSL 端口公开，则应将 EurekaInstanceConfig-nonSecurePortEnabled 中的两个标志更改为 false，将 securePortEnabled 更改为 true。这迫使 Eureka 发布实例信息，显示对安全通信的明确偏好。对于以这种方式配置的服务，Spring Cloud DiscoveryClient 将始终返回以 HTTPS 开头的 URL，并且 Eureka 实例信息将具有安全的运行状况检查 URL。

```
eureka:
 instance:
  nonSecurePortEnabled: false
  securePortEnabled: true
  securePort: ${PORT:8091}
  statusPageUrl: https://localhost:${eureka.instance.securePort}/info
  healthCheckUrl: https://localhost:${eureka.instance.securePort}/health
  homePageUrl: https://localhost:${eureka.instance.securePort}
```

## 12.2.2　通过 HTTPS 服务 Eureka

当 Eureka 服务器以 Spring Boot 启动时，它将部署在嵌入式 Tomcat 容器上，因而 SSL 配置与标准微服务相同。不同之处在于我们必须考虑客户端应用程序，该应用程序将通过 HTTPS 与发现服务器建立安全连接。发现客户端应该针对 Eureka 服务器进行身份验证，并且还应验证服务器的证书。客户端和服务器之间的通信过程称为双向安全套接层（Two-way SSL）或相互身份验证（Mutual Authentication）。还有单向身份验证，这实际上是默认选项，它只有客户端验证服务器的公钥（Public Key）。Java 应用程序使用 KeyStore 和 trustStore 存储与公钥对应的私钥（Private Key）和证书（Certificate）。trustStore

和 KeyStore 之间的唯一区别是它们存储的内容和用途。执行客户端和服务器之间的 SSL 握手时，将使用 trustStore 验证凭据，而使用 KeyStore 提供凭据。换句话说，KeyStore 为给定的应用程序保留私钥和证书，而 trustStore 保留用于从第三方识别它的证书。在配置安全连接时，开发人员通常不会过多关注这些术语，但正确理解它们可以帮助开发人员轻松了解接下来会发生什么。

在典型的基于微服务的架构中，有许多独立的应用程序和单个发现服务器。一方面，每个应用程序都有自己的私钥存储在 KeyStore 中，并且证书对应于 trustStore 中的发现服务器的公钥。另一方面，服务器保留为客户端应用程序生成的所有证书。这就是现在的理论。如图 12.1 所示，它说明了在前面的章节中作为示例使用的系统的情况。

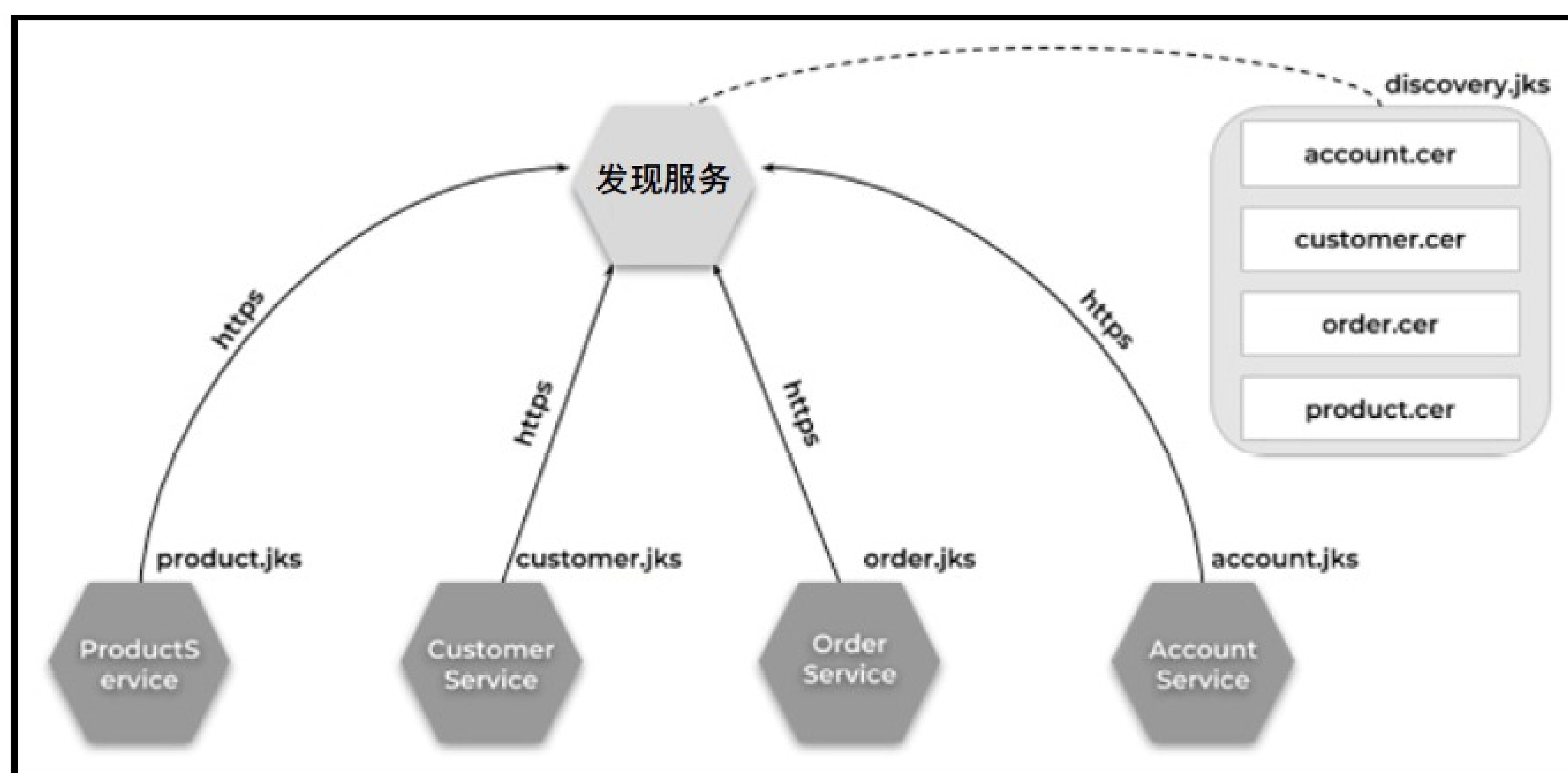



图 12.1　示例系统的安全架构

### 1．Keystore 的生成

在讨论了 Java 中的安全性基础知识之后，即可继续为我们的微服务生成私钥和公钥。和以前一样，我们将使用 JRE 下提供的命令行工具 keytool。让我们从一个众所周知的用于生成带密钥对的 keystore 文件命令开始，首先为发现服务器生成一个 KeyStore，然后为一个选定的微服务（在这个特定示例中，我们选定的是 account-service 服务）生成第二个 KeyStore。

```
keytool -genkey -alias account -store type JKS -keyalg RSA -keysize 2048 -
keystore account.jks -validity 3650
keytool -genkey -alias discovery -storetype JKS -keyalg RSA -keysize 2048 -
keystore discovery.jks -validity 3650
```

然后，必须将自签名证书从 KeyStore 导出到文件，例如，扩展名为.cer 或.crt 的文件。然后，系统将提示输入 KeyStore 生成期间提供的密码。

```
keytool -exportcert -alias account -keystore account.jks -file account.cer
keytool -exportcert -alias discovery -keystore discovery.jks -file
discovery.cer
```

鉴于已经从 KeyStore 中提取了与公钥对应的证书，因而现在可以将其分发给所有感兴趣的各方。来自 account-service 服务的公共证书应包含在发现服务器的 trustStore 中，反之亦然。

```
keytool -importcert -alias discovery -keystore account.jks -file
discovery.cer
keytool -importcert -alias account -keystore discovery.jks -file
account.cer
```

必须为在 Eureka 服务器中注册自身的每个后续微服务重复执行与 account-service 服务相同的步骤。以下是用于为 order-service 服务生成 SSL 密钥和证书的 keytool 命令。

```
keytool -genkey -alias order -storetype JKS -keyalg RSA -keysize 2048 -
keystore order.jks -validity 3650
keytool -exportcert -alias order -keystore order.jks -file order.cer
keytool -importcert -alias discovery -keystore order.jks -file
discovery.cer
keytool -importcert -alias order -keystore discovery.jks -file
order.cer
```

### 2．配置微服务和 Eureka 服务器的 SSL

每个 keystore 文件都放在每个安全微服务和服务发现的 src/main/resources 目录中。每个微服务的 SSL 配置设置与第 12.1 节“为 Spring Boot 启用 HTTPS”中的示例非常相似。唯一的区别是当前使用的 KeyStore 的类型，现在是 JKS 而不是 PKCS12。但是，之前的示例和服务发现配置之间存在更多差异。首先，我们已经通过将 server.ssl.client-auth 属性设置为 need 来启用客户端证书身份验证，这反过来要求我们提供带有 server.ssl.trust-store 属性的 trustStore。以下是 application.yml 文件中发现服务的当前 SSL 配置设置。

```
server:
 port: ${PORT:8761}
 ssl:
  enabled: true
  client-auth: need
  key-store: classpath:discovery.jks
```

```
 key-store-password: 123456
 trust-store: classpath:discovery.jks
 trust-store-password: 123456
 key-alias: discovery
```

如果开发人员使用上述配置运行 Eureka 应用程序，然后尝试访问 https://localhost:8761/ 下可用的Web仪表板，则可能会收到类似SSL_ERROR_BAD_CERT_ALERT的错误代码。发生此错误的原因是没有可信证书导入 Web 浏览器。为此，我们可以从服务（如 account-service 服务）中导入客户端的应用程序 KeyStore。但首先，我们需要将其从 JKS 格式转换为 Web 浏览器支持的另一种格式，如 PKCS12。以下是用于将 KeyStore 从 JKS 转换为 PKCS12 格式的 keytool 命令。

```
keytool -importkeystore -srckeystore account.jks -srcstoretype JKS -
deststoretype PKCS12 -destkeystore account.p12
```

所有最流行的网络浏览器都支持 PKCS12，如 Google Chrome 和 Mozilla Firefox。要将 PKCS12 KeyStore 导入 Google Chrome 浏览器，开发人员可以单击“设置”命令，然后显示“高级”设置部分，找到“管理证书”（管理 HTTPS / SSL 证书和设置），单击打开“证书”对话框。如果再次尝试访问 Eureka Web 仪表板，则应成功进行身份验证，并且将能够看到已注册服务的列表。但是，在这里并没有已注册的应用。为了在发现客户端和服务器之间提供安全通信，我们需要为每个微服务创建一个 DiscoveryClientOptionalArgs 类型的@Bean，它会覆盖发现客户端的实现。有趣的是，Eureka 使用 Jersey 作为 REST 客户端。使用 EurekaJerseyClientBuilder，开发人员可以轻松构建新的客户端实现并传递 keystore 和 truststore 文件的位置。以下是来自 account-service 的代码片段，我们在其中创建了一个新的 EurekaJerseyClient 对象并将其设置为 DiscoveryClientOptionalArgs 的参数。

```
@Bean
public DiscoveryClient.DiscoveryClientOptionalArgs
discoveryClientOptionalArgs() throws NoSuchAlgorithmException {
 DiscoveryClient.DiscoveryClientOptionalArgs args = new
DiscoveryClient.DiscoveryClientOptionalArgs();
 System.setProperty("javax.net.ssl.keyStore",
  "src/main/resources/account.jks");
 System.setProperty("javax.net.ssl.keyStorePassword", "123456");
 System.setProperty("javax.net.ssl.trustStore",
  "src/main/resources/account.jks");
 System.setProperty("javax.net.ssl.trustStorePassword", "123456");
 EurekaJerseyClientBuilder builder = new EurekaJerseyClientBuilder();
 builder.withClientName("account-client");
 builder.withSystemSSLConfiguration();
```

```
    builder.withMaxTotalConnections(10);
    builder.withMaxConnectionsPerHost(10);
    args.setEurekaJerseyClient(builder.build());
    return args;
}
```

在我们的示例系统中，应该为每个微服务提供类似的实现。GitHub（https://github.com/piomin/sample-spring-cloud-security.git）上提供了示例应用程序的源代码。开发人员可以克隆它并使用集成开发环境运行所有 Spring Boot 应用程序。如果一切正常，则应该在 Eureka 仪表板中看到相同的注册服务列表，如图 12.2 所示。如果 SSL 连接有任何问题，则可以尝试在应用程序启动期间设置-Djava.net.debug=ssl VM 参数，以便能够从 SSL 握手过程中检查出完整日志。

Instances currently registered with Eureka

| Application | AMIs | Availability Zones | Status |
|---|---|---|---|
| ACCOUNT-SERVICE | n/a (1) | (1) | UP (1) - minkowp-l.p4.org:account-service:8091 |
| CUSTOMER-SERVICE | n/a (1) | (1) | UP (1) - minkowp-l.p4.org:customer-service:8092 |
| ORDER-SERVICE | n/a (1) | (1) | UP (1) - minkowp-l.p4.org:order-service:8090 |
| PRODUCT-SERVICE | n/a (1) | (1) | UP (1) - minkowp-l.p4.org:product-service:8093 |

图 12.2　在 Eureka 仪表板中看到的注册服务列表

## 12.3　保证配置服务器的安全

在讨论安全性时，我们的架构中还有另外一个关键元素——Spring Cloud Config Server。事实上，保护配置服务器比保护发现服务更重要。为什么？因为我们通常会将它们的身份验证凭据存储到外部系统，甚至还有其他应该隐藏的数据，以防止未经授权的访问和使用。有若干种方法可以正确保护配置服务器。开发人员可以配置 HTTP 基本身份验证、安全 SSL 连接、加密/解密敏感数据，或使用本书第 5 章“使用 Spring Cloud Config 进行分布式配置”中所述的第三方工具。现在来仔细看看其中的一些。

### 12.3.1　加密和解密

在开始之前，我们必须下载并安装 Oracle 提供的 Java Cryptography Extension（JCE）。它由两个 JAR 文件（local_policy.jar 和 US_export_policy.jar）组成，它们需要覆盖 JRE lib/security 目录中的现有策略文件。

如果存储在配置服务器上的远程属性源包含加密数据，则它们的值应以{cipher}为前缀并用引号括起来将其指定为 YAML 文件。.properties 文件不需要包含引号。如果无法解密这样的值，则在前缀为 invalid 的相同键下将其替换为附加值（通常为 <n / a>）。

在上一个示例中，我们存储了用于保护应用程序配置设置中的 keystore 文件的密码。将其保存为纯文本文件可能不是最好的主意，因而它是加密的第一个候选者。问题是，我们应该如何加密它？

幸运的是，Spring Boot 提供了两个可以对此提供帮助的 RESTful 端点。

让我们来看一看它是如何工作的。首先，我们需要启动一个配置服务器实例。要完成该目标，最简单的方法是激活--spring.profiles.active=native 配置文件，该配置文件使用本地类路径或文件系统中的属性源启动服务器。现在我们可以调用两个 POST 端点 /encrypt 和 /decrypt。/encrypt 方法将我们的纯文本密码作为参数。可以使用反向操作 /decrypt 检查出结果，/decrypt 操作会将加密的密码作为参数。

```
$ curl http://localhost:8888/encrypt -d 123456
AQAzI8jv26K3n6ff+iFzQA9DUpWmg79emWu4ndEXyvjYnKFSG7rBmJP0oFTb8RzjZbTwt4
ehRiKWqu5qXkH8SAv/8mr2kdwB28kfVvPj/Lb5hdUkH1TVrylcnpZaKaQYBaxlsa0RWAKQ
Dk8MQKRw1nJ5HM4LY9yjda0YQFNYAy0/KRnwUFihiV5xDk5lMOiG4b77AVLmz+9aSAODKL
O57wOQUzM1tSA7lO9HyDQW2Hzl1q93uOCaP5VQLCJAjmHcHvhlvM442bU3B29JNjH+2nFS
0RhEyUvpUqzo+PBi4RoAKJH9XZ8G7RaTOeWIcJhentKRf0U/EgWIVW21NpsE29BHwf4F2J
ZiWY2+WqcHuHk367X21vk11AVl9tJk9aUVNRk=
```

这里的加密是使用公钥完成的，而解密是使用私钥完成的。因此，如果仅执行加密，则只需要在服务器中提供公钥。出于测试目的，可以使用 keytool 创建 KeyStore。我们之前已经创建了一些 KeyStore，因而不会遇到任何问题。生成的文件应放在类路径中，然后使用 encrypt.keyStore.*属性置于 config-service 配置设置中。

```
encrypt:
 keyStore:
  location: classpath:/config.jks
  password: 123456
  alias: config
  secret: 123456
```

现在，如果将每个微服务的配置设置移动到配置服务器，则可以加密每个密码，如以下示例片段所示。

```
server:
 port: ${PORT:8091}
 ssl:
 enabled: true
```

```
 key-store: classpath:account.jks
 key-store-password:
 '{cipher}AQAzI8jv26K3n6ff+iFzQA9DUpWmg79emWu4ndEXyvjYnKFSG7rBmJP0oFTb
8RzjZbTwt4ehRiKWqu5qXkH8SAv/8mr2kdwB28kfVvPj/Lb5hdUkH1TVrylcnpZaKaQYBa
xlsa0RWAKQDk8MQKRw1nJ5HM4LY9yjda0YQFNYAy0/KRnwUFihiV5xDk5lMOiG4b77AVLm
z+9aSAODKLO57wOQUzM1tSA7lO9HyDQW2Hzl1q93uOCaP5VQLCJAjmHcHvhlvM442bU3B2
9JNjH+2nFS0RhEyUvpUqzo+PBi4RoAKJH9XZ8G7RaTOeWIcJhentKRf0U/EgWIVW21NpsE
29BHwf4F2JZiWY2+WqcHuHk367X21vk11AV19tJk9aUVNRk='
 key-alias: account
```

## 12.3.2　配置客户端和服务器的身份验证

Spring Cloud Config Server 的身份验证实现与 Eureka 服务器完全相同。开发人员可以使用基于标准 Spring 安全机制的 HTTP 基本身份验证。首先，需要确保 spring-security 工件位于类路径中；然后，应该将 security.basic.enabled 设置为 true 以启用安全性，并定义用户名和密码。其示例配置设置如以下代码片段所示。

```
security:
 basic:
  enabled: true
 user:
  name: admin
  password: admin123
```

还必须在客户端启用基本身份验证。它可以通过两种不同的方式实现。第一种方式是通过配置服务器 URL。

```
spring:
 cloud:
  config:
   uri: http://admin:admin123@localhost:8888
```

第二种方法基于单独的 username 和 password 属性。

```
spring:
 cloud:
  config:
   uri: http://localhost:8888
   username: admin
   password: admin123
```

如果要设置 SSL 身份验证，则需要按照第 12.2 节“保证发现服务器的安全”中所描

述的步骤进行操作。使用私钥和证书生成 KeyStore 并设置正确的配置后，开发人员可以运行配置服务器。现在，它将通过 HTTPS 公开其 RESTful API。唯一的区别在于客户端的实现。这是因为 Spring Cloud Config 使用的是与 Spring Cloud Netflix Eureka 不同的 HTTP 客户端。正如你可能猜到的那样，它利用了 RestTemplate，因为它完全是在 Spring Cloud 项目中创建的。

要强制客户端应用程序使用双向 SSL 身份验证而不是标准的非安全 HTTP 连接，首先应该创建一个实现 PropertySourceLocator 接口的@Configuration bean。在这里，我们可以构建一个使用安全 HTTP 连接工厂的自定义 RestTemplate。

```
@Configuration
public class SSLConfigServiceBootstrapConfiguration {

    @Autowired
    ConfigClientProperties properties;

    @Bean
    public ConfigServicePropertySourceLocator
configServicePropertySourceLocator() throws Exception {
        final char[] password = "123456".toCharArray();
        final File keyStoreFile = new
File("src/main/resources/discovery.jks");
        SSLContext sslContext = SSLContexts.custom()
            .loadKeyMaterial(keyStoreFile, password, password)
            .loadTrustMaterial(keyStoreFile).build();
        CloseableHttpClient httpClient =
HttpClients.custom().setSSLContext(sslContext).build();
        HttpComponentsClientHttpRequestFactory requestFactory = new
HttpComponentsClientHttpRequestFactory(httpClient);
        ConfigServicePropertySourceLocator
configServicePropertySourceLocator = new
ConfigServicePropertySourceLocator(properties);
        configServicePropertySourceLocator.setRestTemplate(new
RestTemplate(requestFactory));
        return configServicePropertySourceLocator;
    }

}
```

但是，默认情况下，在应用程序尝试与配置服务器建立连接之前，不会创建此 Bean。

要更改此行为，开发人员还应在 /src/main/resources/META-INF 中创建 spring.factories 文件，并指定自定义引导程序配置类。

```
org.springframework.cloud.bootstrap.BootstrapConfiguration =
pl.piomin.services.account.SSLConfigServiceBootstrapConfiguration
```

## 12.4　使用 OAuth2 进行授权

我们已经在微服务环境中讨论了与身份验证相关的一些概念和解决方案。前文已经演示了微服务和服务发现之间，以及微服务和配置服务器之间的基本和 SSL 身份验证的示例。在服务间通信中，授权似乎比身份验证更重要，而身份验证则在系统的边缘实现。开发人员有必要了解身份验证和授权之间的区别。简而言之，身份验证可以验证访问者的身份，而授权则验证访问者有权执行的操作。

目前，RESTful HTTP API 最流行的授权方法是 OAuth2 和 Java Web 令牌（Java Web Tokens，JWT）。它们可以混合在一起，因为它们比其他解决方案更加互补。Spring 可以为 OAuth 提供商和使用者提供支持。使用 Spring Boot 和 Spring Security OAuth2，开发人员可以快速实现常见的安全模式，如单点登录、令牌中继或令牌交换。但在深入了解有关这些项目的细节以及其他开发细节之前，开发人员需要掌握上述解决方案的基本知识。

### 12.4.1　OAuth2 简介

OAuth2 是几乎所有主要网站目前使用的标准，允许通过共享 API 访问其资源。它将用户身份验证委派给存储用户凭据的独立服务，并授权第三方应用程序访问有关用户账户的共享信息。OAuth2 用于为用户提供数据访问权限，同时保护其账户凭据。它为 Web、桌面和移动应用程序提供流程。以下是与 OAuth2 相关的一些基本术语和角色。

- 资源所有者（Resource Owner）：此角色将控制对资源的访问。此访问权限受到授权范围的限制。
- 授予权限（Authorization Grant）：授予访问权限。可以通过多种方式确认访问，如授权代码、隐式、资源所有者密码凭据和客户端凭据等。
- 资源服务器（Resource Server）：这是一个服务器，用于存储可以使用特殊令牌共享的所有者资源。
- 授权服务器（Authorization Server）：它管理密钥、令牌和其他临时资源访问代码的分配。它还必须确保授予相关用户访问权限。

- 访问令牌（Access Token）：这是允许访问资源的密钥。

为了更好地理解这些术语和角色在实践中的作用，请查看图 12.3。它可视化了使用 OAuth 协议的授权过程的典型流程。

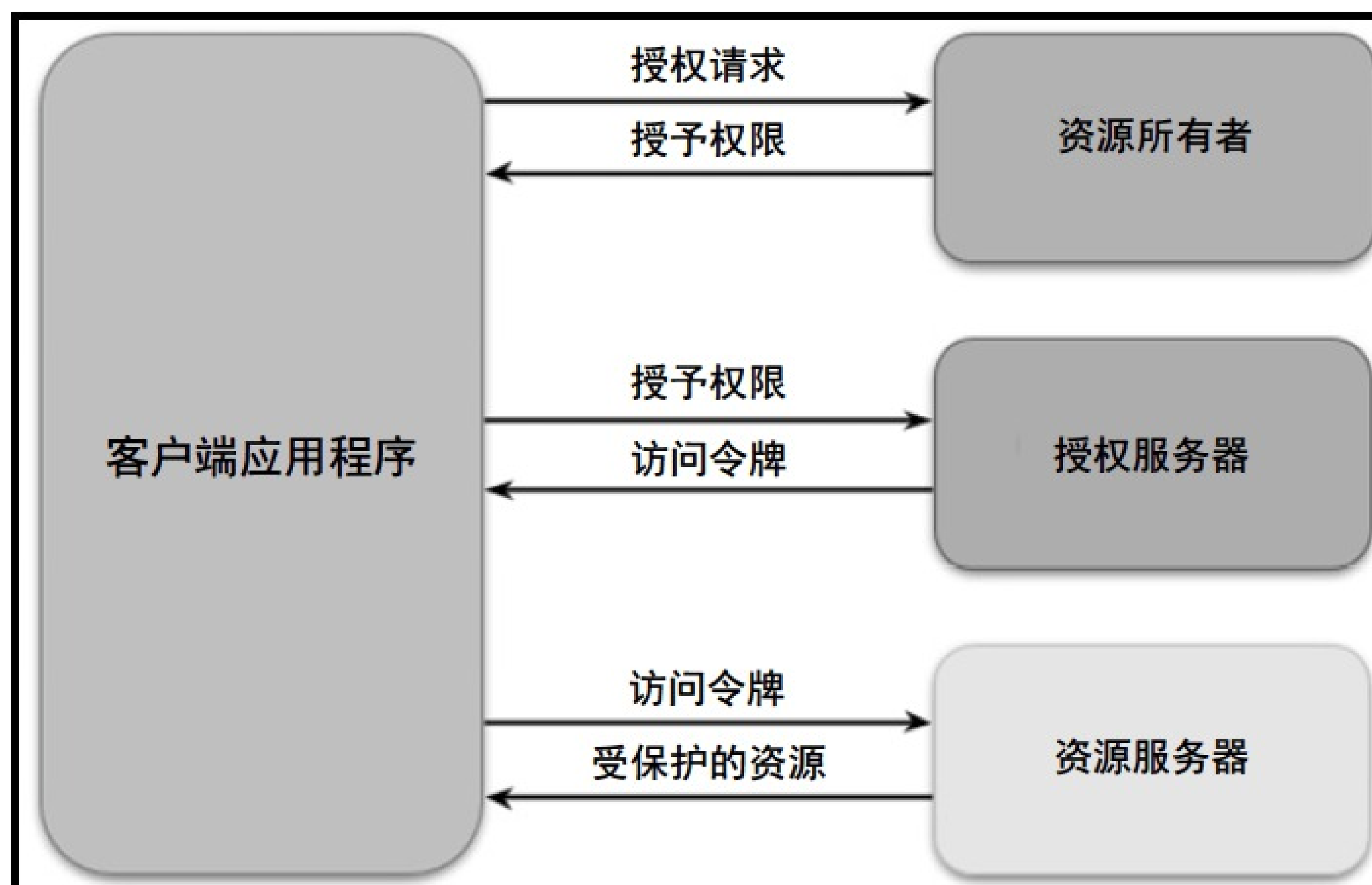



图 12.3　使用 OAuth 协议的授权过程的典型流程

现在我们来看一看之前列出的各个组件之间交互的进一步步骤。应用程序请求资源所有者授权，以便能够访问所请求的服务。资源发送授权作为响应，然后由应用程序将其与其自身的身份一起发送至授权服务器。授权服务器将验证应用程序标识的凭据和授予的权限，然后再发送访问令牌。应用程序使用接收的访问令牌从资源服务器请求资源。最后，如果访问令牌有效，则应用程序能够调用请求服务。

## 12.4.2　构建授权服务器

从单一应用程序迁移到微服务之后，显而易见的解决方案似乎是通过创建授权服务来集中授权工作。使用 Spring Boot 和 Spring Security，开发人员可以轻松创建、配置和启动授权服务器。首先，需要在项目依赖项中包含以下启动器。

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-oauth2</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
```

```
    <artifactId>spring-cloud-starter-security</artifactId>
</dependency>
```

使用 Spring Boot 实现授权服务器模式非常简单，只需要使用@EnableAuthorizationServer 注解主类或配置类，然后在 application.yml 文件中提供 security.oauth2.client.client-id 和 security.oauth2.client.client-secret 属性。当然，这种变体应尽可能简单，因为它定义了客户端详细信息服务的内存实现。

本示例应用程序与本章前面的示例位于同一个存储库（https://github.com/piomin/sample-spring-cloud-security.git）中，但在不同的分支中，即 oauth2 分支（https://github.com/piomin/sample-spring-cloud-security/tree/oauth2）。授权服务器在 auth-service 模块下可用。以下是 auth-service 的 main 类。

```
@SpringBootApplication
@EnableAuthorizationServer
public class AuthApplication {

    public static void main(String[] args) {
        new
SpringApplicationBuilder(AuthApplication.class).web(true).run(args);
    }

}
```

以下是应用程序配置设置的片段。除了客户端的 ID 和机密之外，我们还设置了默认范围并为整个项目启用了基本安全性。

```
security:
  user:
    name: root
    password: password
  oauth2:
    client:
      client-id: piotr.minkowski
      client-secret: 123456
      scope: read
```

运行授权服务之后，我们就可以执行一些测试。例如，可以调用 POST /oauth/token 方法，以便使用资源所有者密码凭据创建访问令牌，就像在以下命令中一样。

```
$ curl piotr.minkowski:123456@localhost:9999/oauth/token -d
grant_type=password -d username=root -d password=password
```

开发人员还可以通过从 Web 浏览器调用 GET /oauth/authorize 端点来使用授权代码授予类型。

```
http://localhost:9999/oauth/authorize?response type=token&
client_id=piotr.minkowski&redirect_uri=http://example.com&scope=read
```

之后，开发人员将被重定向到如图 12.4 所示的批准页面。现在可以确认操作并最终获得访问令牌。它将被发送到初始请求的 redirect_uri 参数中传递的回调 URL。以下是笔者在测试后收到的示例回复。

```
http://example.com/#access token=dd736a4a-1408-4f3f-b3ca-43dcc05e6df0&
token_type=bearer&expires_in=43200.
```

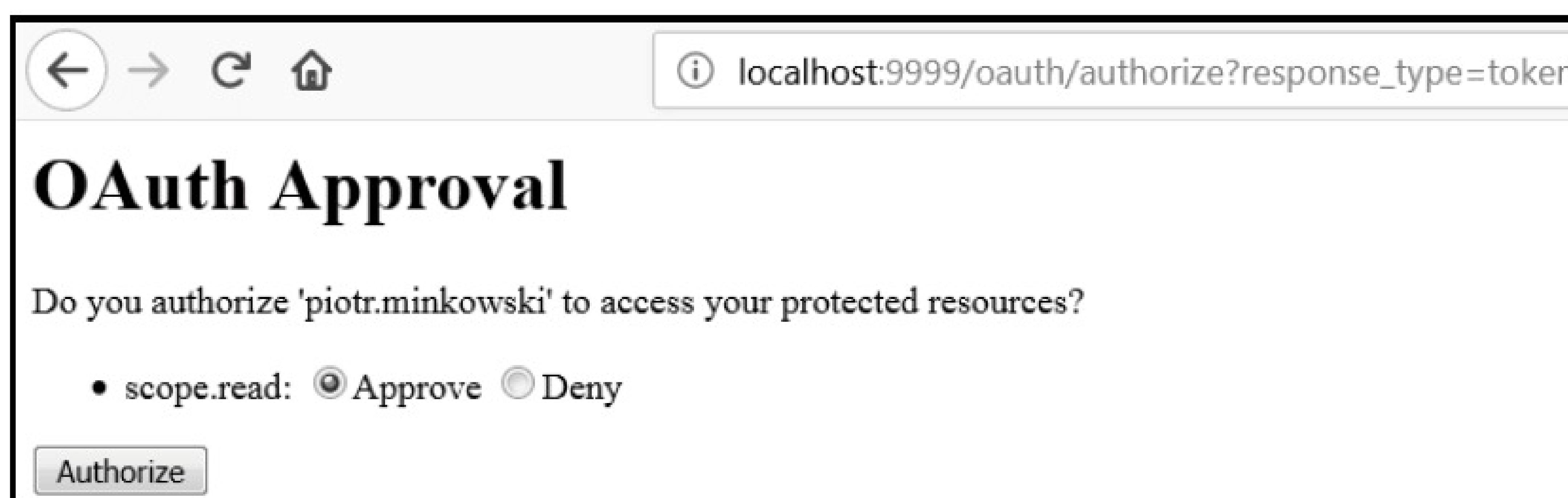



图 12.4　OAuth 批准页面

application.yml 文件中提供的相同 OAuth2 配置也可以按编程的方式实现。为了达到这个目的，开发人员应该声明实现 AuthorizationServerConfigurer 的任何@Beans。其中之一是 AuthorizationServerConfigurerAdapter 适配器，它提供了空方法，允许创建以下独立配置器的不同定义。

- ClientDetailsServiceConfigurer：这定义了客户端详细信息服务。可以初始化客户端详细信息，也可以只引用现有存储。
- AuthorizationServerSecurityConfigurer：这定义了令牌端点 /oauth/token_key 和 /oauth/check_token 的安全约束。
- AuthorizationServerEndpointsConfigurer：这定义了授权和令牌端点以及令牌服务。

这种授权服务器实现的方法为开发人员提供了更多机会。例如，可以使用 ID 和密码定义多个客户端，如以下代码片段所示。下文将介绍一些更高级的示例。

```
@Configuration
@EnableAuthorizationServer
public class AuthServerConfig extends AuthorizationServerConfigurerAdapter
```

```
{

    @Override
    public void configure(AuthorizationServerSecurityConfigurer
oauthServer) throws    Exception {
      oauthServer
        .tokenKeyAccess("permitAll()")
        .checkTokenAccess("isAuthenticated()");
    }

    @Override
    public void configure(ClientDetailsServiceConfigurer clients) throws
Exception {
        clients.inMemory()
            .withClient("piotr.minkowski").secret("123456")
                .scopes("read")
                .authorities("ROLE_CLIENT")
                .authorizedGrantTypes("authorization_code",
"refresh_token", "implicit")
                .autoApprove(true)
            .and()
            .withClient("john.smith").secret("123456")
                .scopes("read", "write")
                .authorities("ROLE_CLIENT")
                .authorizedGrantTypes("authorization_code",
"refresh_token", "implicit")
                .autoApprove(true);
    }
}
```

必须为授权服务器配置的最后一项是 Web 安全性。在扩展 WebSecurityConfigurerAdapter 的类中，我们定义了内存中的用户凭据存储和访问特定资源的权限，如登录页面。

```
@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Autowired
    private AuthenticationManager authenticationManager;

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.requestMatchers()
```

```
            .antMatchers("/login", "/oauth/authorize")
            .and()
            .authorizeRequests()
            .anyRequest().authenticated()
            .and()
            .formLogin().permitAll();
    }

    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws
Exception {
        auth.parentAuthenticationManager(authenticationManager)
            .inMemoryAuthentication()
            .withUser("piotr.minkowski").password("123456")
.roles("USERS");
    }

}
```

### 12.4.3　客户端配置

应用程序可以使用以两种不同方法配置的 OAuth2 客户端。第一种方法是通过@EnableOAuth2Client 注解，它将创建一个 ID 为 oauth2ClientContextFilter 的过滤器 bean，负责存储请求和上下文。它还管理应用程序和授权服务器之间的通信。但是，我们将要讨论的却是第二种方法，即通过@EnableOAuth2Sso 实现 OAuth2 客户端。单点登录（Single Sign-On，SSO）是一种众所周知的安全模式，它允许用户使用一组登录凭据来访问多个应用程序。此注解提供了两个功能——OAuth2 客户端和身份验证。身份验证功能模块使开发人员的应用程序可以符合典型的 Spring Security 机制（如表单登录）。客户端模块则具有与@EnableOAuth2Client 提供的功能相同的特性。所以，开发人员可以将@EnableOAuth2Sso 视为比@EnableOAuth2Client 更高级别的注解。

在下面的示例代码片段中，我们已经注解了使用@EnableOAuth2Sso 扩展 WebSecurityConfigurerAdapter 的类。由于此扩展，Spring Boot 配置了带有 OAuth2 身份验证处理程序的安全过滤器链（Security Filter Chain）。在这种情况下，仅允许对 /login 页面的请求，而所有其他请求都需要身份验证。可以使用 security.oauth2.sso.login-path 属性覆盖表单登录页面路径。在覆盖它之后，开发人员还应该记得在 WebSecurityConfig 中

更改路径模式。

```
@Configuration
@EnableOAuth2Sso
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.antMatcher("/**")
           .authorizeRequests()
           .antMatchers("/login**")
             .permitAll()
           .anyRequest()
             .authenticated();
    }

}
```

还有一些需要设置的配置设置。首先，应该禁用基本身份验证，因为我们已经启用了表单登录方法和@EnableOAuth2Sso 注解。然后，必须提供一些基本的 OAuth2 客户端属性，如客户端凭据和授权服务器公开的 HTTP API 端点的地址。

```
security:
 basic:
  enabled: false
 oauth2:
  client:
   clientId: piotr.minkowski
   clientSecret: 123456
   accessTokenUri: http://localhost:9999/oauth/token
   userAuthorizationUri: http://localhost:9999/oauth/authorize
  resource:
   userInfoUri: http://localhost:9999/user
```

application.yml 文件片段中的最后一个属性是 security.oauth2.resource.userInfoUri，它需要服务器端的其他端点。通过 UserController 实现的端点将返回 java.security.Principal 对象，指示当前经过身份验证的用户。

```
@RestController
public class UserController {

    @RequestMapping("/user")
    public Principal user(Principal user) {
```

```
        return user;
    }

}
```

现在，如果调用由我们的某个微服务公开的任何端点，都将自动重定向到登录页面。由于我们为内存客户端的详细信息存储设置了 autoApprove 选项，因而将自动生成授予的权限和访问令牌，而无须用户进行任何交互。在登录页面提供凭据后，开发人员应该从请求的资源处获得响应。

### 12.4.4　使用 JDBC 后端存储

在前面的小节中，我们配置了一个身份验证服务器和客户端应用程序，它授予对资源服务器进行保护的资源的访问权限。但是，整个授权服务器配置已在内存中提供。这种解决方案在开发过程中满足了我们的需求，但在生产模式中却不是最理想的方法。目标解决方案应将所有身份验证凭据和令牌存储在数据库中。开发人员可以在 Spring 支持的许多关系数据库之间进行选择。在本示例中，我们决定使用 MySQL。

因此，第一步就是在本地启动 MySQL 数据库。实现这一目标的最简便方式是通过 Docker 容器。除了启动数据库之外，以下命令还会创建一个模式（Schema）和一个名为 oauth2 的用户。

```
docker run -d --name mysql -e MYSQL_DATABASE=oauth2 -e MYSQL_USER=oauth2 -e
MYSQL_PASSWORD=oauth2 -e MYSQL_ALLOW_EMPTY_PASSWORD=yes -p 33306:3306 mysql
```

一旦启动了 MySQL，现在必须在客户端提供连接设置。如果开发人员是在 Windows 机器和端口 33306 上运行 Docker，则 MySQL 在主机地址 192.168.99.100 下可用。数据源属性应在 auth-service 的 application.yml 文件中设置。Spring Boot 还能够在应用程序启动时在所选数据源上运行一些 SQL 脚本。这对开发人员来说是一个好消息，因为我们必须在专用于 OAuth2 流程的架构上创建一些表。

```
spring:
 application:
  name: auth-service
 datasource:
  url: jdbc:mysql://192.168.99.100:33306/oauth2?useSSL=false
  username: oauth2
  password: oauth2
  driver-class-name: com.mysql.jdbc.Driver
```

```
  schema: classpath:/script/schema.sql
  data: classpath:/script/data.sql
```

已创建的模式包含一些用于存储 OAuth2 凭据和令牌的表，如 tokens-oauth_client_details、oauth_client_token、oauth_access_token、oauth_refresh_token、oauth_code 和 oauth_approvals。此外，在/src/main/resources/script/schema.sql 中还提供了带有 SQL 创建命令的完整脚本。

实际上，还有第二个 SQL 脚本（/src/main/resources/script/data.sql），它带有一些用于测试用途的 insert 命令。最重要的是添加一些客户端 ID/客户端密钥对。

```
INSERT INTO `oauth_client_details` (`client_id`, `client_secret`, `scope`,
`authorized_grant_types`, `access_token_validity`,
`additional_information`) VALUES ('piotr.minkowski', '123456', 'read',
'authorization_code,password,refresh_token,implicit', '900', '{}');
INSERT INTO `oauth_client_details` (`client_id`, `client_secret`, `scope`,
`authorized_grant_types`, `access_token_validity`,
`additional_information`) VALUES ('john.smith', '123456', 'write',
'authorization_code,password,refresh_token,implicit', '900', '{}');
```

当前版本的身份验证服务器与基本示例中描述的版本之间存在一些实现上的差异。这里最重要的事情是将默认令牌存储设置为数据库，方法是提供一个以默认数据源作为参数的 JdbcTokenStore bean。虽然所有令牌现在都存储在数据库中，但我们仍然希望以 JWT 格式生成它们。这就是为什么必须在该类中提供第二个 bean JwtAccessTokenConverter 的原因。通过覆盖从基类继承的不同 configure 方法，开发人员可以为 OAuth2 客户端详细信息设置默认存储，并将授权服务器配置为始终验证在 HTTP 标头中提交的 API 密钥。

```
@Configuration
@EnableAuthorizationServer
public class OAuth2Config extends AuthorizationServerConfigurerAdapter {

    @Autowired
    private DataSource dataSource;
    @Autowired
    private AuthenticationManager authenticationManager;

    @Override
    public void configure(AuthorizationServerEndpointsConfigurer endpoints)
throws Exception {
        endpoints.authenticationManager(this.authenticationManager)
            .tokenStore(tokenStore())
            .accessTokenConverter(accessTokenConverter());
```

```
    }

    @Override
    public void configure(AuthorizationServerSecurityConfigurer
oauthServer) throws Exception {
        oauthServer.checkTokenAccess("permitAll()");
    }

    @Bean
    public JwtAccessTokenConverter accessTokenConverter() {
        return new JwtAccessTokenConverter();
    }

    @Override
    public void configure(ClientDetailsServiceConfigurer clients) throws
Exception {
        clients.jdbc(dataSource);
    }

    @Bean
    public JdbcTokenStore tokenStore() {
        return new JdbcTokenStore(dataSource);
    }

}
```

Spring 应用程序可以提供自定义的身份验证机制。要在应用程序中使用它，必须实现 UserDetailsService 接口并覆盖其 loadUserByUsername 方法。在我们的示例应用程序中，用户凭据和权限也存储在数据库中，因而需要将 UserRepository bean 注入自定义 UserDetailsService 类。

```
@Component("userDetailsService")
public class UserDetailsServiceImpl implements UserDetailsService {

    private final Logger log =
LoggerFactory.getLogger(UserDetailsServiceImpl.class);

    @Autowired
    private UserRepository userRepository;

    @Override
    @Transactional
```

```
    public UserDetails loadUserByUsername(final String login) {
        log.debug("Authenticating {}", login);
        String lowercaseLogin = login.toLowerCase();
        User userFromDatabase;
        if(lowercaseLogin.contains("@")) {
            userFromDatabase = userRepository.findByEmail(lowercaseLogin);
        } else {
            userFromDatabase =
userRepository.findByUsernameCaseInsensitive(lowercaseLogin);
        }
        if (userFromDatabase == null) {
            throw new UsernameNotFoundException("User " + lowercaseLogin +
" was not found in the database");
        } else if (!userFromDatabase.isActivated()) {
            throw new UserNotActivatedException("User " + lowercaseLogin +
" is not activated");
        }
        Collection<GrantedAuthority> grantedAuthorities = new
ArrayList<>();
        for (Authority authority : userFromDatabase.getAuthorities()) {
            GrantedAuthority grantedAuthority = new
SimpleGrantedAuthority(authority.getName());
            grantedAuthorities.add(grantedAuthority);
        }
        return new
org.springframework.security.core.userdetails.User(userFromDatabase.
getUser name(), userFromDatabase.getPassword(), grantedAuthorities);
    }

}
```

## 12.4.5　服务间授权

使用 Feign 客户端即可实现本示例中的服务间通信。以下是我们所选的实现之一（这里选择的是本示例中的 order-service 服务），它将调用来自 customer-service 服务的端点。

```
@FeignClient(name = "customer-service")
public interface CustomerClient {

    @GetMapping("/withAccounts/{customerId}")
    Customer findByIdWithAccounts(@PathVariable("customerId") Long
```

```
customerId);

}
```

与其他服务一样，来自 customer-service 服务的所有可用方法都受到基于 OAuth 令牌范围的预授权机制的保护。它允许开发人员使用@PreAuthorize 注解每个方法，定义所需的范围。

```
@PreAuthorize("#oauth2.hasScope('write')")
@PutMapping
public Customer update(@RequestBody Customer customer) {
    return repository.update(customer);
}

@PreAuthorize("#oauth2.hasScope('read')")
@GetMapping("/withAccounts/{id}")
public Customer findByIdWithAccounts(@PathVariable("id") Long id) throws
JsonProcessingException {
    List<Account> accounts = accountClient.findByCustomer(id);
    LOGGER.info("Accounts found: {}", mapper.writeValueAsString(accounts));
    Customer c = repository.findById(id);
    c.setAccounts(accounts);
    return c;
}
```

默认情况下预授权机制是被禁用的。要为 API 方法启用它，应该使用@EnableGlobalMethodSecurity 批注。开发人员还应该指出这样的预授权将基于 OAuth2 令牌范围。

```
@Configuration
@EnableResourceServer
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class OAuth2ResourceServerConfig extends
GlobalMethodSecurityConfiguration {

    @Override
    protected MethodSecurityExpressionHandler createExpressionHandler() {
        return new OAuth2MethodSecurityExpressionHandler();
    }

}
```

如果通过 Feign 客户端调用 account-service 服务端点，则会出现以下异常。

```
feign.FeignException: status 401 reading
CustomerClient#findByIdWithAccounts();
content:{"error":"unauthorized","error_description":
"Full authentication is required to access this resource"}
```

为什么会出现这种异常呢？当然，customer-service 服务受 OAuth2 令牌授权保护，但 Feign 客户端不会在请求标头中发送授权令牌。可以通过为 Feign 客户端定义自定义配置类来自定义该方法。它允许开发人员声明一个请求拦截器。在这种情况下，可以使用来自 Spring Cloud OAuth2 库的 OAuth2FeignRequestInterceptor 提供的 OAuth2 实现。出于测试目的，笔者决定使用资源所有者密码授予类型。

```
public class CustomerClientConfiguration {

    @Value("${security.oauth2.client.access-token-uri}")
    private String accessTokenUri;
    @Value("${security.oauth2.client.client-id}")
    private String clientId;
    @Value("${security.oauth2.client.client-secret}")
    private String clientSecret;
    @Value("${security.oauth2.client.scope}")
    private String scope;

    @Bean
    RequestInterceptor oauth2FeignRequestInterceptor() {
        return new OAuth2FeignRequestInterceptor(new
DefaultOAuth2ClientContext(), resource());
    }

    @Bean
    Logger.Level feignLoggerLevel() {
        return Logger.Level.FULL;
    }

    private OAuth2ProtectedResourceDetails resource() {
        ResourceOwnerPasswordResourceDetails resourceDetails = new
ResourceOwnerPasswordResourceDetails();
        resourceDetails.setUsername("root");
        resourceDetails.setPassword("password");
        resourceDetails.setAccessTokenUri(accessTokenUri);
        resourceDetails.setClientId(clientId);
        resourceDetails.setClientSecret(clientSecret);
        resourceDetails.setGrantType("password");
```

```
        resourceDetails.setScope(Arrays.asList(scope));
        return resourceDetails;
    }

}
```

最后，我们可以测试已经实现的解决方案。这一次，我们将创建一个 JUnit 自动化测试，而不是在 Web 浏览器中单击它或使用其他工具发送请求。测试方法显示在以下代码段中。开发人员可以使用 OAuth2RestTemplate 和 ResourceOwnerPasswordResourceDetails 来执行资源所有者凭据授权操作，并使用请求标头中发送的 OAuth2 令牌从 order-service 服务调用 POST / API 方法。当然，在运行该测试之前，必须启动所有微服务，以及发现和授权服务器。

```
@Test
public void testClient() {
    ResourceOwnerPasswordResourceDetails resourceDetails = new
ResourceOwnerPasswordResourceDetails();
    resourceDetails.setUsername("root");
    resourceDetails.setPassword("password");
    resourceDetails.setAccessTokenUri("http://localhost:9999/oauth/token");
    resourceDetails.setClientId("piotr.minkowski");
    resourceDetails.setClientSecret("123456");
    resourceDetails.setGrantType("password");
    resourceDetails.setScope(Arrays.asList("read"));
    DefaultOAuth2ClientContext clientContext = new
DefaultOAuth2ClientContext();
    OAuth2RestTemplate restTemplate = new
OAuth2RestTemplate(resourceDetails, clientContext);
    restTemplate.setMessageConverters(Arrays.asList(new
MappingJackson2HttpMessageConverter()));
    Random r = new Random();
    Order order = new Order();
    order.setCustomerId((long) r.nextInt(3) + 1);
    order.setProductIds(Arrays.asList(new Long[] {
(long) r.nextInt(10) + 1, (long) r.nextInt(10) + 1 }));
    order = restTemplate.postForObject("http://localhost:8090", order,
Order.class);
    if (order.getStatus() != OrderStatus.REJECTED) {
        restTemplate.put("http://localhost:8090/{id}", null,
order.getId());
    }
}
```

## 12.4.6　在 API 网关上启用 SSO

开发人员可以通过使用@EnableOAuth2Sso 注解 main 类来启用 API 网关上的单点登录功能。实际上，这是微服务架构的最佳选择，它可以强制使 Zuul 为当前经过身份验证的用户生成或获取访问令牌。

```
@SpringBootApplication
@EnableOAuth2Sso
@EnableZuulProxy
public class GatewayApplication {

    public static void main(String[] args) {
        new
SpringApplicationBuilder(GatewayApplication.class).web(true).run(args);
    }

}
```

通过包含@EnableOAuth2Sso，可以触发 ZuulFilter 的自动配置。过滤器负责从当前经过身份验证的用户提取访问令牌，然后将其放入请求标头中，并转发到隐藏在网关后面的微服务中。如果为这些服务激活了@EnableResourceServer，它们将在 Authorization HTTP 标头中接收预期的令牌。可以通过声明 proxy.auth.*属性来控制@EnableZuulProxy下游的授权行为。

在架构中使用网关时，可能会隐藏其后面的授权服务器。在这种情况下，应该在 Zuul 的配置设置中提供其他路由，如 uaa。然后，OAuth2 客户端和服务器之间交换的所有消息都将通过网关。以下是网关的 application.yml 文件中的正确配置。

```
security:
  oauth2:
    client:
      accessTokenUri: /uaa/oauth/token
      userAuthorizationUri: /uaa/oauth/authorize
      clientId: piotr.minkowski
      clientSecret: 123456
    resource:
      userInfoUri: http://localhost:9999/user

zuul:
  routes:
```

```
    account-service:
      path: /account/**
    customer-service:
      path: /customer/**
    order-service:
      path: /order/**
    product-service:
      path: /product/**
    uaa:
      sensitiveHeaders:
      path: /uaa/**
      url: http://localhost:9999
  add-proxy-headers: true
```

## 12.5　小　　结

本章是专门开辟的一个关于安全主题的章节，以按步骤详细介绍如何保护基于微服务的架构的关键元素。与安全相关的主题通常比其他主题更高级，因此，笔者花了一些时间来解释该领域的若干基本概念。本章通过示例说明了双向 SSL 身份验证、敏感数据的加密/解密、Spring Security 身份验证以及使用 JWT 令牌的 OAuth2 授权。至于在架构中应该使用哪些组件才能提供所需的安全级别，将由开发人员自己来决定。

在阅读完本章之后，开发人员应该能够为应用程序设置基本和更高级的安全配置。此外，还应该能够保护系统架构的每个组件。当然，本章只讨论了一些可能的解决方案和框架。例如，不必仅依赖 Spring 作为授权服务器提供程序，我们也可能会使用第三方工具，如 Keycloak，它可以充当基于微服务的系统中的授权和身份验证服务器。它还可以很轻松地与 Spring Boot 应用程序集成。它支持所有最流行的协议，如 OAuth2、OpenId Connect 和 SAML。事实上，Keycloak 是一个非常强大的工具，应该被视为 Spring 授权服务器的替代品，特别是对于大型企业系统和其他更高级的用例而言更是如此。

第 13 章将讨论微服务测试的不同策略。

# 第 13 章　测试 Java 微服务

在开发新应用程序时，我们也不应该忘记自动化测试。如果考虑使用基于微服务的架构，这些则特别重要。测试微服务所需要采用的方法与测试一体化应用程序所采用的方法不同。就一体化应用程序而言，主要关注的是单元测试和集成测试，以及数据库层。而在微服务的情况下，最重要的是以尽可能最细的粒度为每个通信提供覆盖。虽然每个微服务都是独立开发和发布的，但其中一个微服务的更改可能会影响与该服务交互的所有其他服务。它们之间的通信是通过消息实现的。一般来说，这些是通过 REST 或 AMQP 协议发送的消息。

本章将要讨论的主题包括：

- Spring 的自动化测试支持。
- Spring Boot 微服务的组件和集成测试之间的差异。
- 使用 Pact 实现契约测试。
- 使用 Spring Cloud Contract 实现契约测试。
- 使用 Gatling 实现性能测试。

## 13.1　测试策略

有 5 种不同的微服务测试策略，前 3 个策略与一体化应用程序相同。

- 单元测试（Unit Test）：通过单元测试，开发人员可以测试最小的代码片段，例如，单个方法或组件，并模拟其他方法和组件的每次调用。有许多流行的框架均支持 Java 中的单元测试，如 JUnit、TestNG 和 Mockito（用于模拟）。此类测试的主要任务是确认实现符合要求。单元测试可以是一个强大的工具，尤其是与测试驱动开发模式相结合时。
- 集成测试（Integration Test）：仅使用单元测试并不能保证开发人员可以验证整个系统的行为。集成测试采用模块并尝试一起测试它们。这种方法使开发人员有机会在子系统中测试通信路径。我们将测试组件之间的交互和通信，这些组件基于与模拟的外部服务的接口。在基于微服务的系统中，可以使用集成测试以包括其他微服务、数据源或高速缓存。
- 端到端测试（End-to-End Test）：端到端测试也称功能测试（Function Test）。

这些测试的主要目标是验证系统是否满足外部要求。这意味着我们应该设计测试方案来测试参与该过程的所有微服务。良好的端到端测试设计并非易事。由于我们需要测试整个系统，因而特别强调测试的方案设计非常重要。

- 契约测试（Contract Test）：契约测试用于确保微服务的显式和隐式契约按预期工作。当使用者与组件的接口集成以便使用它时，总是形成契约。通常，在基于微服务的系统中，存在单个组件的许多使用者。他们每个人通常都需要一份满足其要求的不同契约。遵循这些假设，每个使用者都负责源组件的接口行为。
- 组件测试（Component Test）：在完成微服务中所有对象和方法的单元测试之后，开发人员应该单独测试整个微服务。为了单独运行测试，开发人员需要模拟（Mock）或桩（Stub）其他微服务的调用。外部数据存储应替换为等效的内存数据存储，这也提供了显著的测试性能改进。

契约测试和组件测试之间的差异是显而易见的。图 13.1 说明了我们的示例 order-service 微服务中契约测试和组件测试之间的差异。

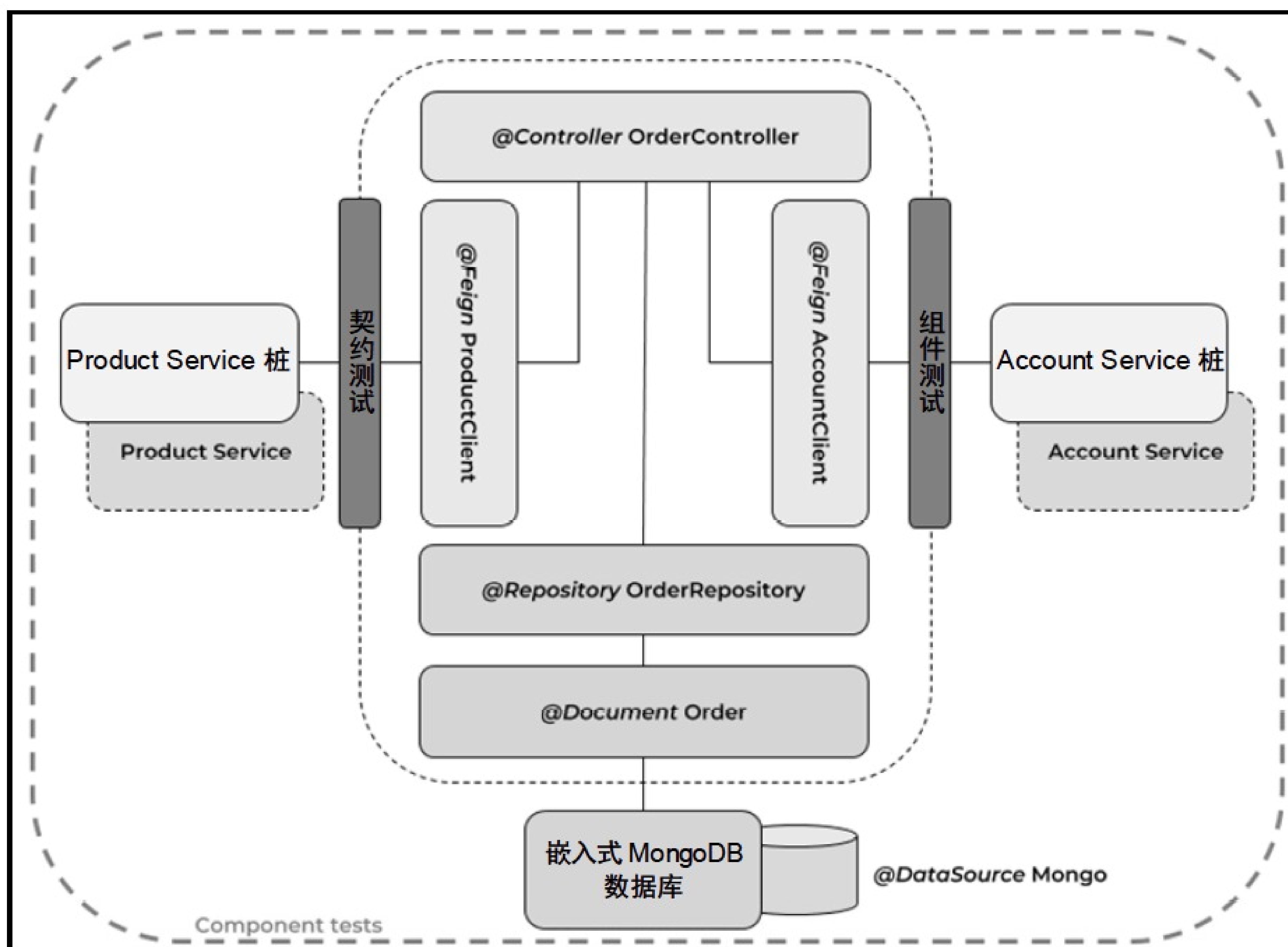



图 13.1　契约测试和组件测试之间的差异

现在，有一个问题是：我们是否确实需要两个额外的策略来测试基于微服务的系统？通过适当的单元和集成测试，开发人员就可以确认组成微服务的各个组件的实现是否正确。但是，如果没有针对微服务的更具体的测试策略，开发人员将无法确定它们如何协同工作以满足业务需求。

因此，我们增加了组件测试和契约测试。这是一个非常重要的变化，它可以帮助我们理解组件测试、契约测试和集成测试之间的差异。由于组件测试是与外界隔离进行的，因而集成测试将负责验证与该世界的交互。这就是为什么我们应该为组件测试提供桩以进行集成测试。契约测试很像集成测试，强调微服务之间的交互，但是契约测试会将微服务视为黑盒（Black Box）并仅验证响应的格式。

一旦为微服务提供功能测试，开发人员还应该考虑性能测试（Performance Test）。我们将区分以下性能测试策略。

- 负载测试（Load Test）：这些测试用于确定系统在正常和预期负载条件下的行为。这里的主要思想是识别一些弱点，如响应时间延迟、异常中断，或者如果未正确设置网络超时，则重试次数过多等。
- 压力测试（Stress Test）：这些测试将检测系统的上限，以检查它在极重负载下的行为。除负载测试外，它还检查内存泄漏、安全问题和数据损坏。它可能使用与负载测试相同的工具。

图 13.2 说明了在系统上执行所有测试策略的逻辑顺序。我们将从最简单的单元测试开始，它将验证软件的各个小段，然后逐步经过下一个阶段，直至最终完成压力测试，将整个系统推向极限。

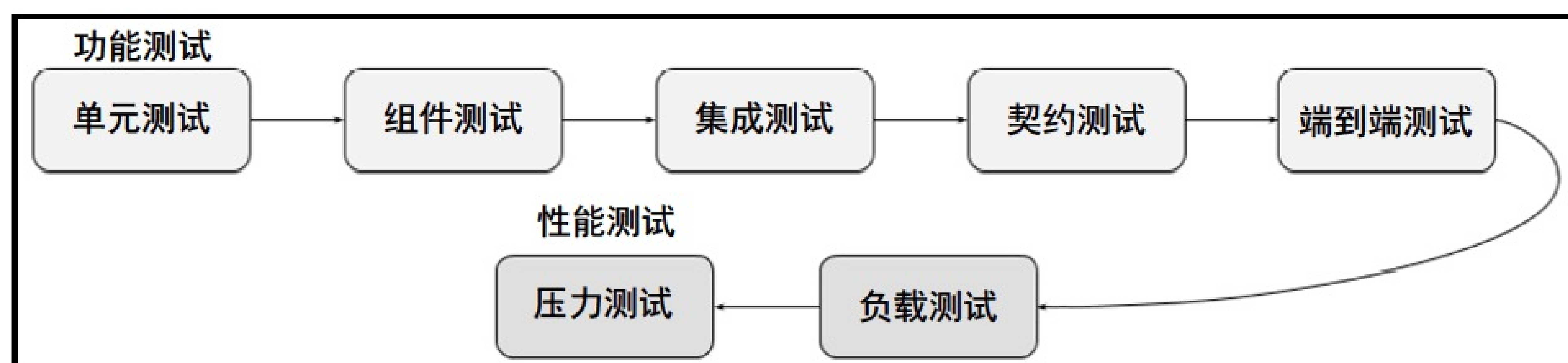



图 13.2　在系统上执行所有测试策略的逻辑顺序

## 13.2　测试 Spring Boot 应用程序

正如第 13.1 节所述，在应用程序测试中有一些不同的测试策略和方法。前文已经简要提到了所有这些理论知识，所以现在我们可以进入实际环节。Spring Boot 提供了一组

有助于实现自动化测试的实用程序。为了在项目中启用这些功能，必须将 spring-boot-starter-test 启动器包含在依赖项中。它不仅会导入 spring-test 和 spring-boot-test 工件，还会导入一些其他有用的测试库，如 JUnit、Mockito 和 AssertJ。

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
</dependency>
```

## 13.2.1　构建示例应用程序

在开始进行自动化测试之前，需要出于测试目的而准备一个示例业务逻辑。我们可以使用本书前面章节中的相同示例系统，但必须对其进行一些修改。到目前为止，我们从未使用外部数据源来存储和收集测试数据。在本章中，为了说明不同策略如何处理持久性测试问题，则有必要这样做。现在，每个服务都应该有自己的数据库，当然，在通常情况下，选择哪个数据库并不重要。Spring Boot 支持多种解决方案，包括关系数据库和 NoSQL 数据库。我们决定使用的数据库是 Mongo。我们先来了解一下示例系统的架构。图 13.3 显示的当前示例系统架构的模型就已经考虑了在每个服务中都采用专用数据库的假设。

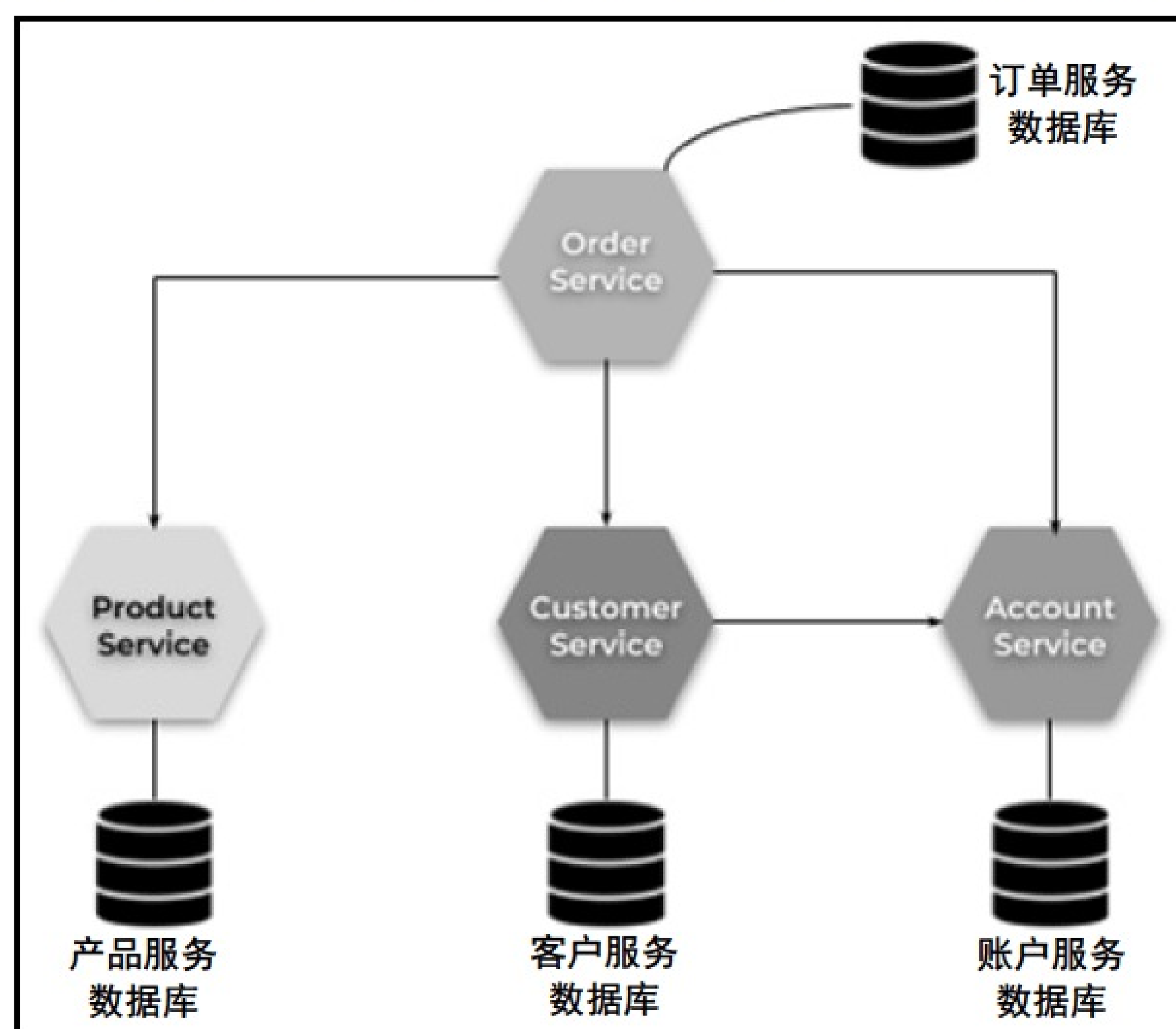



图 13.3　当前示例系统架构的模型

## 13.2.2　与数据库集成

为了对我们的 Spring Boot 应用程序启用 Mongo 支持，可以将 spring-boot-starter-data-mongo 启动器包含在依赖项中。该项目提供了一些有趣的功能来简化与 MongoDB 的集成。在这些功能中，值得一提的是特定的富对象映射（Rich Object Mapping）——MongoTemplate，当然还有对其他 Spring Data 项目众所周知的存储库编写风格的支持。以下是 pom.xml 中所需的依赖项声明。

```
<dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-data-mongodb</artifactId>
</dependency>
```

可以使用 Docker 镜像轻松启动 MongoDB 的实例。运行以下命令即可启动 Docker 容器并在端口 27017 上公开 Mongo 数据库。

```
docker run --name mongo -p 27017:27017 -d mongo
```

为了将应用程序与先前启动的数据源连接，开发人员应该覆盖 application.yml 中的一些自动配置设置。这可以通过 spring.data.mongodb.*属性来实现。

```
spring:
 application:
  name: account-service
 data:
  mongodb:
   host: 192.168.99.100
   port: 27017
   database: micro
   username: micro
   password: micro123
```

前文已经提到了对象映射功能。Spring Data Mongo 提供了一些可用于此的注解。存储在数据库中的每个对象都应该使用@Document 注解。目标集合的主键是一个 12 字节的字符串，应该使用 Spring Data @Id 在每个映射的类中指示。以下是 Account 对象实现的代码片段。

```
@Document
public class Account {

    @Id
```

```
    private String id;
    private String number;
    private int balance;
    private String customerId;
    // ...

}
```

## 13.3　单 元 测 试

前面花了较多时间来描述与 MongoDB 的集成。但是，测试持久性是自动化测试的关键点之一，因而正确配置它非常重要。现在，我们可以继续进行测试的实现。Spring Test 可以为最典型的测试方案提供支持，如通过 REST 客户端与其他服务集成或与数据库集成。我们有一组库可以让开发人员轻松模拟与外部服务的交互，这对于单元测试来说尤为重要。

以下测试类是 Spring Boot 应用程序的典型单元测试实现。我们使用了 JUnit 框架，它是 Java 的事实标准。这里使用 Mockito 库来替换真实的存储库和控制器及其桩。这种方法使我们可以轻松验证@Controller 类实现的每个方法的正确性。测试与外部组件隔离进行，这是单元测试的主要假设。

```
@RunWith(SpringRunner.class)
@WebMvcTest(AccountController.class)
public class AccountControllerUnitTest {

    ObjectMapper mapper = new ObjectMapper();

    @Autowired
    MockMvc mvc;
    @MockBean
    AccountRepository repository;

    @Test
    public void testAdd() throws Exception {
        Account account = new Account("1234567890", 5000, "1");
        when(repository.save(Mockito.any(Account.class))).thenReturn(new
Account("1","1234567890", 5000, "1"));
mvc.perform(post("/").contentType(MediaType.APPLICATION JSON).
content(mappe r.writeValueAsString(account)))
```

```
.andExpect(status().isOk());
}

    @Test
    public void testWithdraw() throws Exception {
        Account account = new Account("1", "1234567890", 5000, "1");
        when(repository.findOne("1")).thenReturn(account);
        when(repository.save(Mockito.any(Account.class))).thenAnswer(new
Answer<Account>() {
            @Override
            public Account answer(InvocationOnMock invocation) throws
Throwable {
                Account a = invocation.getArgumentAt(0, Account.class);
                return a;
            }
        });
        mvc.perform(put("/withdraw/1/1000"))
            .andExpect(status().isOk())
.andExpect(content().contentType(MediaType.APPLICATION_JSON_UTF8))
            .andExpect(jsonPath("$.balance", is(4000)));
    }

}
```

好消息是，开发人员可以轻松地模拟 Feign 客户端通信（特别是在微服务环境中）。以下示例测试类将通过调用 account-service 服务公开的端点来验证用于提取资金的 order-service 服务的端点。你可能已经注意到，该端点又由先前引入的测试类进行了测试。以下是 order-service 服务的单元测试实现的类。

```
@RunWith(SpringRunner.class)
@WebMvcTest(OrderController.class)
public class OrderControllerTest {

    @Autowired
    MockMvc mvc;
    @MockBean
    OrderRepository repository;
    @MockBean
    AccountClient accountClient;

    @Test
    public void testAccept() throws Exception {
```

```
        Order order = new Order("1", OrderStatus.ACCEPTED, 2000, "1", "1",
null);
        when(repository.findOne("1")).thenReturn(order);
        when(accountClient.withdraw(order.getAccountId(),
order.getPrice())).thenReturn(new Account("1", "123", 0));
        when(repository.save(Mockito.any(Order.class))).thenAnswer(new
Answer<Order>() {
            @Override
            public Order answer(InvocationOnMock invocation) throws
Throwable {
                Order o = invocation.getArgumentAt(0, Order.class);
                return o;
            }
        });

        mvc.perform(put("/1"))
            .andExpect(status().isOk())
.andExpect(content().contentType(MediaType.APPLICATION JSON UTF8))
            .andExpect(jsonPath("$.status", is("DONE")));
    }

}
```

## 13.4　组件测试

如果已为应用程序中的所有关键类和接口提供了单元测试，则可以继续进行组件测试。组件测试的主要思想是使用内存中测试双精度和数据存储来实例化内存中的完整微服务。这允许我们跳过网络连接。对于单元测试，我们模拟的是所有数据库或 HTTP 客户端，而在组件测试中我们则不用模拟任何东西。我们将为数据库客户端提供内存数据源，并模拟 REST 客户端的 HTTP 响应。

### 13.4.1　使用内存数据库运行测试

我们选择 MongoDB 的原因之一是它可以很容易地嵌入 Spring Boot 应用程序以进行测试。要为项目启用嵌入式 MongoDB，可以在 Maven 的 pom.xml 文件中包含以下依赖项。

```
<dependency>
    <groupId>de.flapdoodle.embed</groupId>
```

```
    <artifactId>de.flapdoodle.embed.mongo</artifactId>
    <scope>test</scope>
</dependency>
```

Spring Boot 可以为嵌入式 MongoDB 提供自动配置，因此，除了在 application.yml 中设置本地地址和端口之外，我们不需要做任何其他事情。

因为，在默认情况下，我们将使用在 Docker 容器上运行的 Mongo，我们应该在另一个 Spring 配置文件中声明这样的配置。通过使用@ActiveProfiles 注解测试类，在测试用例执行期间激活此特定配置文件。以下是 application.yml 的一个片段，我们使用不同的 MongoDB 连接设置定义了两个配置文件——dev 和 test。

```
---
spring:
 profiles: dev
 data:
  mongodb:
   host: 192.168.99.100
   port: 27017
   database: micro
   username: micro
   password: micro123

---
spring:
 profiles: test
 data:
  mongodb:
   host: localhost
   port: 27017
```

如果使用 MongoDB 以外的数据库，如 MySQL 或 Postgres，则可以使用替代的内存中（In-Memory）嵌入式关系数据库（如 H2 或 Derby）轻松替换它们。Spring Boot 支持它们，并将为可能使用@DataJpaTest 激活的测试提供自动配置。开发人员也可以使用@DataMongoTest 注解来嵌入 MongoDB，而不是使用@SpringBootTest。除了内存中的嵌入式 MongoDB，它还将配置 MongoTemplate，扫描@Document 类，并配置 Spring Data MongoDB 存储库。

### 13.4.2　处理 HTTP 客户端和服务发现

虽然关于使用内存中数据库测试持久性的问题已得到解决，但是，我们仍需要考虑

测试的其他方面，如模拟来自其他服务的 HTTP 响应或与服务发现集成。当开发人员为微服务实现某些测试时，可以选择两种典型的服务发现方法。第一个是在测试用例执行期间将发现服务器嵌入应用程序，第二个是在客户端禁用发现。使用 Spring Cloud 配置第二个选项相对容易。对于 Eureka Server 来说，可以使用 eureka.client.enabled = false 属性禁用它。

这只是练习的第一部分。我们还应该禁用 Ribbon 客户端的发现，该客户端负责在服务间通信中进行负载均衡。如果有多个目标服务，则必须使用服务名称标记每个客户端。以下配置中的最后一个属性 listOfServers 的值与用于自动化测试实现的框架严格相关。接下来我们将演示基于 Hoverfly Java 库的示例（该库在本书第 7 章“高级负载均衡和断路器”中已经介绍过），然后使用它来模拟调用目标服务的延迟，以便显示 Ribbon 客户端和 Hystrix 如何处理网络超时。在这里，我们将使用它来返回准备好的响应，以使我们的组件测试触及网络通信。以下是配置文件的一个片段，其中的配置选项将负责禁用 Eureka 的发现，并设置 Ribbon 客户端的测试属性。此外，还应该通过使用@ActiveProfiles 注解来为测试类激活该配置文件。

```
---
spring:
 profiles: no-discovery
eureka:
 client:
  enabled: false
account-service:
 ribbon:
  eureka:
   enable: false
  listOfServers: account-service:8080
customer-service:
 ribbon:
  eureka:
   enable: false
  listOfServers: customer-service:8080
product-service:
 ribbon:
  eureka:
   enable: false
  listOfServers: product-service:8080
```

我们不想深入探讨 Hoverfly 的使用细节，因为本书第 7 章“高级负载均衡和断路器”对其已经介绍过。如前文所述，可以通过声明使用 HoverflyRule 的@ClassRule 来激活

Hoverfly 以进行 JUnit 测试，定义应该模拟的服务和端点列表。每个服务的名称必须与使用 listOfServers 属性定义的地址相同。以下是 Hoverfly 测试规则的定义，该规则将模拟来自 3 种不同服务的响应。

```
@ClassRule
public static HoverflyRule hoverflyRule = HoverflyRule
 .inSimulationMode(dsl(
 service("account-service:8080")
 .put(startsWith("/withdraw/"))
.willReturn(success("{\"id\":\"1\",\"number\":\"1234567890\",
\"balance\":5000}", "application/json")),
 service("customer-service:8080")
 .get("/withAccounts/1")
 .willReturn(success("{\"id\":\"{{ Request.Path.[1]
}}\",\"name\":\"Test1\",\"type\":\"REGULAR\",\"accounts\":[{\"id\":\"1\",
\" number\":\"1234567890\",\"balance\":5000}]}", "application/json")),
 service("product-service:8080")
 .post("/ids").anyBody()
 .willReturn(success("[{\"id\":\"1\",\"name\":\"Test1\",
\"price\":1000}]", "application/json"))))
 .printSimulationData();
```

## 13.4.3　实现示例测试

为了验证在上两节中已经提出过的结论，现在可以使用内存中嵌入式 MongoDB、Hoverfly（模拟 HTTP 响应）和禁用服务发现来准备组件测试。我们还特别为测试目的而准备了可在 test 和 no-discovery 配置项下使用的正确配置设置。每个组件测试都将由 TestRestTemplate 初始化，TestRestTemplate 将调用 order-service HTTP 端点。可以基于 HTTP 响应或存储在嵌入式 MongoDB 中的数据来执行测试结果验证。以下是 order-service 服务的组件测试的示例实现。

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
@FixMethodOrder(MethodSorters.NAME_ASCENDING)
@ActiveProfiles({"test", "no-discovery"})
public class OrderComponentTest {

    @Autowired
    TestRestTemplate restTemplate;
    @Autowired
```

```
    OrderRepository orderRepository;
    // ...
    @Test
    public void testAccept() {
        Order order = new Order(null, OrderStatus.ACCEPTED, 1000, "1", "1",
Collections.singletonList("1"));
        order = orderRepository.save(order);
        restTemplate.put("/{id}", null, order.getId());
        order = orderRepository.findOne(order.getId());
        Assert.assertEquals(OrderStatus.DONE, order.getStatus());
    }

    @Test
    public void testPrepare() {
        Order order = new Order(null, OrderStatus.NEW, 1000, "1", "1",
Collections.singletonList("1"));
        order = restTemplate.postForObject("/", order, Order.class);
        Assert.assertNotNull(order);
        Assert.assertEquals(OrderStatus.ACCEPTED, order.getStatus());
        Assert.assertEquals(940, order.getPrice());
    }

}
```

## 13.5　集 成 测 试

在创建单元和组件测试之后，我们已经验证了微服务中的所有功能。但是，我们仍然需要测试与其他服务、外部数据存储和缓存的交互。在基于微服务的架构集成中，测试的处理方式与一体化应用程序中的处理方式不同。由于内部模块之间的所有关系都已通过组件测试方法进行了测试，因而我们仅需要测试与外部组件交互的模块。

### 13.5.1　对测试进行分类

在持续集成（Continuous Integration，CI）管道中分离集成测试也是有意义的，这样外部中断不会阻止或破坏项目的构建。开发人员应该考虑通过使用@Category 对它们进行注解来对测试进行分类。可以创建特别用于集成测试的接口，如 IntegrationTest。

```
public interface IntegrationTest  { }
```

然后，可以使用@Category 注解在该接口上标记测试。

```
@Category(IntegrationTest.class)
public class OrderIntegrationTest { ... }
```

最后，可以将 Maven 配置为仅运行所选类型的测试，如使用 maven-failsafe-plugin。

```
<plugin>
  <artifactId>maven-failsafe-plugin</artifactId>
  <dependencies>
    <dependency>
      <groupId>org.apache.maven.surefire</groupId>
      <artifactId>surefire-junit47</artifactId>
    </dependency>
  </dependencies>
  <configuration>
    <groups>pl.piomin.services.order.IntegrationTest</groups>
  </configuration>
  <executions>
    <execution>
      <goals>
        <goal>integration-test</goal>
      </goals>
      <configuration>
        <includes>
          <include>**/*.class</include>
        </includes>
      </configuration>
    </execution>
  </executions>
</plugin>
```

### 13.5.2　捕获 HTTP 流量

分类是在自动化测试期间处理与外部微服务通信问题的方法之一。解决该问题的另一种流行方法是记录传出请求和传入响应，以便将来使用它们而无须建立与外部服务的连接。

在前面的示例中，我们仅在模拟模式下使用 Hoverfly。但是，它也可以在捕获模式下运行，这意味着它将正常地向真实服务发出请求，但是它们将被 Hoverfly 拦截、记录并存储在文件中。然后，可以在模拟模式中使用以 JSON 格式存储的已捕获流量的文件。开发人员可以在 JUnit 测试类中创建 Hoverfly 规则，如果模拟文件不存在，则以捕获模式

启动；如果模拟文件存在，则以模拟模式启动。它始终存储在 src/test/resources/hoverfly 目录中。

这就是打破外部服务依赖关系的简单方法。例如，如果开发人员知道没有更改，则无须与实际服务进行交互。如果要修改此类服务，则可以删除 JSON 模拟文件，从而切换到捕获模式。如果测试失败，则意味着修改会影响到服务，开发人员必须在返回捕获模式之前执行一些修复。

以下是位于 order-service 服务中的集成测试示例。它会添加一个新账户，然后调用从该账户中提取资金的方法。由于使用了 inCaptureOrSimulationMode 方法，仅当 account.json 文件不存在或开发人员更改了传递给测试中的服务的输入数据时，才会调用实际服务。

```
@RunWith(SpringRunner.class)
@SpringBootTest
@ActiveProfiles("dev")
@Category(IntegrationTest.class)
public class OrderIntegrationTest {

    @Autowired
    AccountClient accountClient;
    @Autowired

    CustomerClient customerClient;
    @Autowired
    ProductClient productClient;
    @Autowired
    OrderRepository orderRepository;

    @ClassRule
    public static HoverflyRule hoverflyRule =
HoverflyRule.inCaptureOrSimulationMode("account.json").printSimulationData();

    @Test
    public void testAccount() {
        Account account = accountClient.add(new Account(null, "123",
5000));
        account = accountClient.withdraw(account.getId(), 1000);
        Assert.notNull(account);
        Assert.equals(account.getBalance(), 4000);
    }

}
```

# 13.6　契 约 测 试

有一些有趣的工具专门用于契约测试。我们将通过两个最流行的工具——Pact 和 Spring Cloud Contract 来讨论这个概念。

## 13.6.1　使用 Pact

如前文所述，契约测试的主要概念是定义使用者（Consumer）和提供者（Provider）之间的契约，然后针对每个服务独立地进行验证。由于创建和维护契约的责任主要在于使用者端，因而这种类型的测试通常被称为使用者驱动的测试（Consumer-Driven Test）。在 Pact JVM 中可以清楚地看到对使用者和提供者方面的划分。它提供了两个独立的库，第一个以 pact-jvm-consumer 为前缀，第二个以 pact-jvm-provider 为前缀。当然，契约是由使用者与提供者协商创建的，如图 13.4 所示。

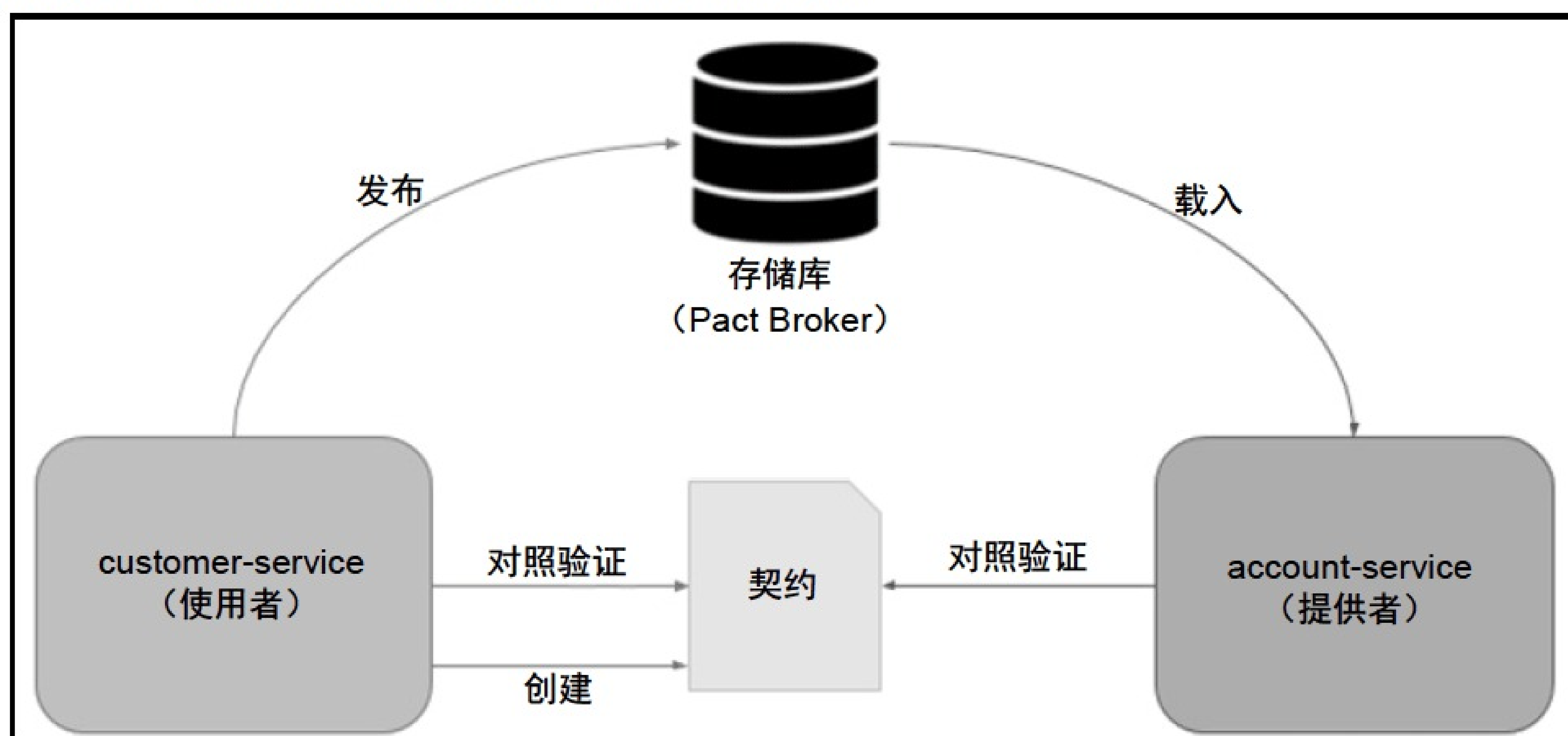



图 13.4　契约测试示意图

事实上，Pact 是一系列框架，可以为使用者驱动的契约测试提供支持。这些实现可用于不同的语言和框架。幸运的是，Pact 可以与 JUnit 和 Spring Boot 一起使用。现在可以来考虑在我们的示例系统中实现的集成之一，即 customer-service 服务和 account-service 服务之间的集成。名为 customer-service 服务的微服务将使用 Feign 客户端与 account-service 服务进行通信。使用者端的 Feign 客户定义事实上代表了我们的契约。

```
@FeignClient(name = "account-service")
public interface AccountClient {

    @GetMapping("/customer/{customerId}")
    List<Account> findByCustomer(@PathVariable("customerId")
String customerId);

}
```

### 1. 使用者端

要在使用者端启用具有 JUnit 支持的 Pact，可以在项目中包含以下依赖项。

```
<dependency>
    <groupId>au.com.dius</groupId>
    <artifactId>pact-jvm-consumer-junit 2.12</artifactId>
    <version>3.5.12</version>
    <scope>test</scope>
</dependency>
```

现在我们唯一要做的就是创建 JUnit 测试类。可以通过使用@SpringBootTest 注解并使用 Spring Runner 运行它来将其实现为标准的 Spring Boot 测试。要成功执行创建的测试，我们首先需要禁用发现客户端，并确保 Ribbon 客户端将与@Rule PactProviderRuleMk2 所代表的 account-service 服务的桩通信。测试的关键点是 callAccountClient 方法，该方法使用@Pact 进行注解并返回 RequestResponsePact。它定义了请求的格式和响应的内容。在测试用例执行期间，Pact 会自动生成该定义的 JSON 表示，它在 target/pacts/addressClient-customerServiceProvider.json 文件中可用。最后，调用 Feign 客户端中实现的方法，并在使用@PactVerification 注解的测试方法中验证 Pact @Rule 返回的响应。以下是一个 customer-service 服务的使用者端契约测试的实现示例。

```
@RunWith(SpringRunner.class)
@SpringBootTest(properties = {
 "account-service.ribbon.listOfServers: localhost:8092",
 "account-service.ribbon.eureka.enabled: false",
 "eureka.client.enabled: false",
})
public class CustomerConsumerContractTest {

    @Rule
    public PactProviderRuleMk2 stubProvider = new
PactProviderRuleMk2("customerServiceProvider", "localhost", 8092, this);
```

```
    @Autowired
    private AccountClient accountClient;

    @Pact(state = "list-of-3-accounts", provider =
"customerServiceProvider", consumer = "accountClient")
    public RequestResponsePact callAccountClient(PactDslWithProvider
builder) {
        return builder.given("list-of-3-accounts").uponReceiving("test-
account-service")
.path("/customer/1").method("GET").willRespondWith().status(200)
.body("[{\"id\":\"1\",\"number\":\"123\",\"balance\":5000},{\"id\":\"2\",
\"number\":\"124\",\"balance\":5000},{\"id\":\"3\",\"number\":\"125\",
\"balan ce\":5000}]", "application/json").toPact();
    }

    @Test
    @PactVerification(fragment = "callAccountClient")
    public void verifyAddressCollectionPact() {
        List<Account> accounts = accountClient.findByCustomer("1");
        Assert.assertEquals(3, accounts.size());
    }

}
```

在 target/pacts 目录中生成的 JSON 测试结果文件必须在提供者端可用。最简单的解决方案假设它只能使用@PactFolder 注解访问生成的文件。当然，它要求提供者可以访问 target/pacts 目录。虽然它应该可以用于我们的示例，因为它的源代码存储在同一个 Git 存储库中，但它并不是我们的目标解决方案。幸运的是，我们可以使用 Pact Broker 在网络中发布 Pact 测试结果。Pact Broker 是一个存储库服务器，它可以提供用于发布和使用 Pact 文件的 HTTP API。

开发人员可以使用 Docker 镜像在本地启动 Pact Broker。它需要 Postgres 数据库作为后端存储，因而也需要启动带 Postgres 的容器。以下是所需的 Docker 命令。

```
docker run -d --name postgres -p 5432:5432 -e POSTGRES_USER=oauth -e
POSTGRES_PASSWORD=oauth123 -e POSTGRES_DB=oauth postgres
docker run -d --name pact-broker --link postgres:postgres -e
PACT_BROKER_DATABASE_USERNAME=oauth -e
PACT_BROKER_DATABASE_PASSWORD=oauth123 -e
PACT_BROKER_DATABASE_HOST=postgres -e PACT_BROKER_DATABASE_NAME=oauth -p
9080:80 dius/pact_broker
```

在 Docker 上运行 Pact Broker 之后，必须在那里发布我们的测试报告。可以使用 Maven 插件 pact-jvm-provider-maven_2.12 轻松执行此操作。如果运行 mvn clean install pack:publish 命令，则放置在/target/pacts 目录中的所有文件都将被发送到代理的 HTTP API。

```
<plugin>
    <groupId>au.com.dius</groupId>
    <artifactId>pact-jvm-provider-maven_2.12</artifactId>
    <version>3.5.12</version>
    <configuration>
        <pactBrokerUrl>http://192.168.99.100:9080</pactBrokerUrl>
    </configuration>
</plugin>
```

可以使用 http://192.168.99.100:9080 上提供的 Web 控制台显示已发布的 Pact 的完整列表。它还提供有关上次验证日期的信息以及列表中每个 Pact 的详细信息，如图 13.5 所示即为其屏幕截图。

Pacts

| Consumer | | Provider | Latest pact published | Webhook status | Last verified |
|---|---|---|---|---|---|
| accountClient | | customerServiceProvider | 1 day ago | Create | about 1 hour ago |
| accountClient | | orderServiceProvider | 1 minute ago | Create | |

2 pacts

图 13.5　已发布的 Pact 的完整列表

## 2．生产者端

假设使用者创建了一个 Pact 并将其发布在代理上，则开发人员可以继续在提供者端实现验证测试。要在提供者端启用具有 JUnit 支持的 Pact，可以在项目中包含 pact-jvm-provider-junit 依赖项。

还有另一个框架也可以使用，即 pact-jvm-provider-spring。该库允许开发人员使用 Spring 和 JUnit 对提供者运行契约测试。可以在 Maven 的 pom.xml 的以下片段中看到所需依赖项的列表。

```
<dependency>
    <groupId>au.com.dius</groupId>
    <artifactId>pact-jvm-provider-junit_2.12</artifactId>
    <version>3.5.12</version>
    <scope>test</scope>
```

```
</dependency>
<dependency>
    <groupId>au.com.dius</groupId>
    <artifactId>pact-jvm-provider-spring_2.12</artifactId>
    <version>3.5.12</version>
    <scope>test</scope>
</dependency>
```

由于 Spring 专用库的存在，开发人员可以使用 SpringRestPactRunner 而不是默认的 PactRunner。反过来，这也允许开发人员使用 Spring 测试注解，如@MockBean。在下面的 JUnit 测试中，我们模拟了 AccountRepository bean。它将返回使用者端测试所需的 3 个对象。测试将自动启动 Spring Boot 应用程序并调用 /customer/{customerId}端点。另外还有两件重要的事情。通过使用@Provider 和@State 注解，我们需要设置与@Pact 注解中使用者端的测试设置相同的名称。最后，通过在测试类上声明@PactBroker，可以为 Pact 的存储库提供连接设置。以下是使用 Pact 的测试示例，它将验证由 customer-service 服务发布的契约。

```
@RunWith(SpringRestPactRunner.class)
@Provider("customerServiceProvider")
@PactBroker(host = "192.168.99.100", port = "9080")
@SpringBootTest(webEnvironment =
SpringBootTest.WebEnvironment.DEFINED_PORT, properties = {
"eureka.client.enabled: false" })
public class AccountProviderContractTest {

    @MockBean
    private AccountRepository repository;
    @TestTarget
    public final Target target = new HttpTarget(8091);

    @State("list-of-3-accounts")
    public void toDefaultState() {
        List<Account> accounts = new ArrayList<>();
        accounts.add(new Account("1", "123", 5000, "1"));
        accounts.add(new Account("2", "124", 5000, "1"));
        accounts.add(new Account("3", "125", 5000, "1"));
        when(repository.findByCustomerId("1")).thenReturn(accounts);
    }

}
```

### 13.6.2　使用 Spring Cloud Contract

Spring Cloud Contract 提供的契约测试方法与 Pack 略有不同。在 Pack 中，使用者负责发布契约，在 Spring Cloud Contract 中，此操作的发起者却是提供者。契约作为 JAR 存储在 Maven 存储库中，包含基于契约定义文件自动生成的桩。可以使用 Groovy DSL 语法创建这些定义。它们中的每一个都包含两个主要部分：请求和响应规范。在这些文件的基础上，Spring Cloud Contract 生成 JSON 桩定义，WireMock 使用它们在客户端进行集成测试。与 Pact（用作支持 REST API 的使用者驱动的契约测试的工具）相比，它专门用于测试基于 JVM 的微服务。它由以下 3 个子项目组成。

- Spring Cloud Contract Verifier
- Spring Cloud Contract Stub Runner
- Spring Cloud Contract WireMock

现在我们不妨来分析如何在契约测试中使用它们，为方便起见，测试仍然使用之前在第 13.6.1 节“使用 Pact”中描述的相同示例。

**注意：**

WireMock 是基于 HTTP 的 API 的模拟器。有些开发人员可能会认为它是服务虚拟化工具或模拟服务器。通过捕获流入和流出现有 API 的流量，它能够快速启动和运行。

#### 1．定义契约和生成桩

如前文所述，与 Pact 相反，在 Spring Cloud Contract 中，提供者（服务器端）负责发布契约规范。因此，我们将从 account-service 服务开始实现，该服务为 customer-service 服务调用的端点提供服务。但在继续实现之前，请看图 13.6 中的图解，它说明了参与测试过程的主要组件。

示例应用程序的源代码在与先前示例相同的 GitHub 存储库中可用，但在不同的 contract 分支上。

要为提供者端应用程序启用 Spring Cloud Contract 功能，首先必须将 Spring Cloud Contract Verifier 包含在项目依赖项中。

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-contract-verifier</artifactId>
    <scope>test</scope>
</dependency>
```

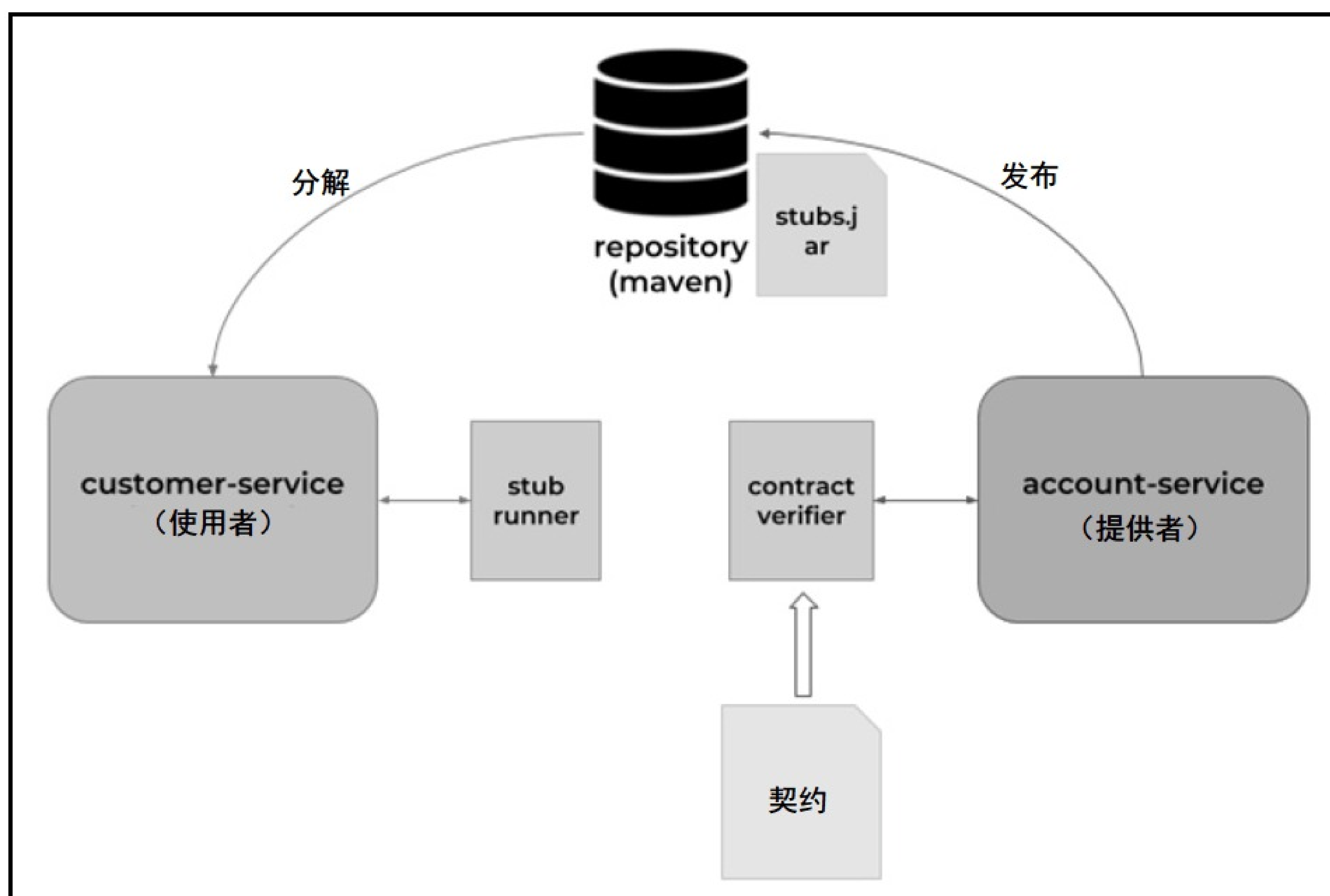



图 13.6　参与测试过程的主要组件

下一步是添加 Spring Cloud Contract Verifier Maven 插件，该插件将生成并运行契约测试。它还会在本地 Maven 存储库中生成和安装桩。它有一个必须定义的唯一参数，即由生成的测试类扩展的基类所在的包（Package）。

```
<plugin>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-contract-maven-plugin</artifactId>
    <version>1.2.0.RELEASE</version>
    <extensions>true</extensions>
    <configuration>
<packageWithBaseClasses>pl.piomin.services.account</packageWithBaseClasses>
    </configuration>
</plugin>
```

现在，我们必须为契约测试创建一个基类。它应该放在 pl.piomin.services.account 包中。在下面的基类中，将使用@SpringBootTest 设置 Spring Boot 应用程序，然后模拟 AccountRepository。开发人员还可以使用 RestAssured 来模拟 Spring MVC，并仅向控制器

发送请求。由于上述模拟的存在，测试不会与任何外部组件（如数据库或 HTTP 端点）交互，并仅测试契约。

```
@RunWith(SpringRunner.class)
@SpringBootTest(classes = {AccountApplication.class})
public abstract class AccountProviderTestBase {

    @Autowired
    private WebApplicationContext context;
    @MockBean
    private AccountRepository repository;

    @Before
    public void setup() {
        RestAssuredMockMvc.webAppContextSetup(context);
        List<Account> accounts = new ArrayList<>();
        accounts.add(new Account("1", "123", 5000, "1"));
        accounts.add(new Account("2", "124", 5000, "1"));
        accounts.add(new Account("3", "125", 5000, "1"));
        when(repository.findByCustomerId("1")).thenReturn(accounts);
    }

}
```

我们已经提供了使用 Spring Cloud Contract 运行测试所需的所有配置和基类。因此，现在可以进入最重要的部分，使用 Spring Cloud Contract Groovy DSL 定义契约。契约的所有规范都应位于/src/test/resources/contracts 目录中。此目录下的特定位置（包含桩定义）将被视为基本测试类名。每个桩定义代表单个契约测试。根据此规则，spring-cloud-contract-maven-plugin 会自动查找契约并将其分配给基础测试类。在目前讨论的示例中，我们将桩定义放在/src/test/resources/contracts/accountService 目录中。因此，生成的测试类名称为 AccountServiceTest，它还扩展了 AccountServiceBase 类。

以下是契约规范的示例，它返回属于客户的账户列表。这份契约不是很简单，所以有些事情需要解释。开发人员可以使用正则表达式在 Contract DSL 中编写请求。还可以根据通信方（使用者或生产者）为每个属性提供不同的值。Contract DSL 还使开发人员能够使用 fromRequest 方法在响应中引用请求。以下契约将返回 3 个账户的列表，从请求路径和 id 字段中获取 customerId 字段，其中包含 5 个数字。

```
org.springframework.cloud.contract.spec.Contract.make {
 request {
  method 'GET'
```

```
  url value(consumer(regex('/customer/[0-9]{3}')), producer('/customer/1'))
 }
 response {
  status 200
  body([
   [
    id: $(regex('[0-9]{5}')),
    number: '123',
    balance: 5000,
    customerId: fromRequest().path(1)
   ], [
    id: $(regex('[0-9]{5}')),
    number: '124',
    balance: 5000,
    customerId: fromRequest().path(1)
   ], [
    id: $(regex('[0-9]{5}')),
    number: '125',
    balance: 5000,
    customerId: fromRequest().path(1)
   ]
  ])
  headers {
   contentType(applicationJson())
  }
 }
}
```

在 Maven 构建的测试阶段，在 target/generated-test-sources 目录下将生成测试类。以下是从上述契约规范中生成的类。

```
public class AccountServiceTest extends AccountServiceBase {

    @Test
    public void validate_customerContract() throws Exception {

        // given:
        MockMvcRequestSpecification request = given();

        // when:
        ResponseOptions response = given().spec(request)
.get("/customer/1");
```

```
        // then:
        assertThat(response.statusCode()).isEqualTo(200);
        assertThat(response.header("Content-
Type")).matches("application/json.*");

        // and:
        DocumentContext parsedJson =
JsonPath.parse(response.getBody().asString());
assertThatJson(parsedJson).array().contains("['number']").isEqualTo("123");
assertThatJson(parsedJson).array().contains("['balance']").isEqualTo(5000);
assertThatJson(parsedJson).array().contains("['number']").isEqualTo("124");
assertThatJson(parsedJson).array().contains("['customerId']").isEqualTo("1");
assertThatJson(parsedJson).array().contains("['id']").matches("[0-9]{5}");
    }

}
```

### 2．验证使用者方面的契约

假设我们已经在提供者端成功构建并运行了测试，那么将生成桩，然后在本地 Maven 存储库中发布。为了能够在使用者应用程序测试期间使用它们，应该将 Spring Cloud Contract Stub Runner 包含到项目依赖项中。

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-contract-stub-runner</artifactId>
    <scope>test</scope>
</dependency>
```

然后，我们应该用@AutoConfigureStubRunner 注解测试类。它需要两个输入参数——ids 和 workOffline。ids 字段是 artifactId、groupId、版本号、stubs 限定符和端口号的连接产物，并且通常指向提供者发布的桩的 JAR。workOffline 标志指示具有桩的存储库所在的位置。默认情况下，使用者将尝试从 Nexus 或 Artifactory 自动下载工件。如果开发人员希望强制 Spring Cloud Contract Stub Runner 仅从本地 Maven 存储库下载桩，则可以将 workOffline 参数的值切换为 true。

以下是一个 JUnit 测试类，它使用 Feign 客户端从提供者端发布的桩中调用端点。Spring Cloud Contract 会查找 pl.piomin.services:account-service 工件的最新版本。已经通过在@AutoConfigureStubRunner 注解中将 + 符号作为桩的版本传递来表示。如果开发人员想要使用该工件的具体版本，则可以从 pom.xml 文件而不是使用 + 符号来设置当前版本，如@AutoConfigureStubRunner(ids={"pl.piomin.services:account-service:1.0-SNAPSHOT:

stubs:8091" })。

```
@RunWith(SpringRunner.class)
@SpringBootTest(properties = {
 "eureka.client.enabled: false"
})
@AutoConfigureStubRunner(ids = {"pl.piomin.services:account-
service:+:stubs:8091"}, workOffline = true)
public class AccountContractTest {

    @Autowired
    private AccountClient accountClient;

    @Test
    public void verifyAccounts() {
        List<Account> accounts = accountClient.findByCustomer("1");
        Assert.assertEquals(3, accounts.size());
    }

}
```

现在剩下的唯一事情是使用 mvn clean install 命令构建整个项目，以验证测试是否成功运行。但是，开发人员应该记住的是，之前创建的测试仅涵盖 customer-service 服务和 account-service 服务之间的集成。在我们的示例系统中，应该验证的微服务之间还有一些其他的集成。我们将演示一个测试整个系统的示例。它将测试已公开的 order-service 服务的方法，并且该服务将与所有其他微服务进行通信。为此，我们将使用 Spring Cloud Contract 方案的另一个有趣功能。

### 3. 方案

使用 Spring Cloud Contract 定义方案（Scenario）并不困难。唯一需要做的就是在创建契约时提供正确的命名约定。此约定假定作为方案一部分的每个契约的名称都以订单号和下画线为前缀。单个方案中包含的所有契约都必须位于同一目录中。Spring Cloud Contract 方案基于 WireMock 的方案。以下是一个目录结构，其中包含为创建和接受订单的方案需求定义的合约。

```
src\main\resources\contracts
 orderService\
  1_createOrder.groovy
  2_acceptOrder.groovy
```

以下是为该方案生成的测试的源代码。

```
@FixMethodOrder(MethodSorters.NAME ASCENDING)
public class OrderScenarioTest extends OrderScenarioBase {

    @Test
    public void validate 1 createOrder() throws Exception {
        // ...
    }

    @Test
    public void validate 2 acceptOrder() throws Exception {
        // ...
    }

}
```

现在，假设我们有很多微服务，并且大多数微服务与一个或多个其他微服务进行通信。因此，即使测试单个契约，也无法确保在服务间通信期间所有其他契约按预期工作。但是，使用 Spring Cloud Contract，则可以轻松地将所有必需的桩包含在测试类中，这使得开发人员能够验证已定义方案中的所有契约。这需要包含 spring-cloud-starter-contract-verifier 和 spring-cloud-starter-contract-stub-runner 依赖项。以下类定义将充当 Spring Cloud Contract 测试类的基础，并包括由其他微服务生成的桩。为 order-service 服务端点生成的桩可以由需要使用 order-service 服务验证契约的任何其他外部服务使用。像以下代码这样的测试不仅将验证此服务与 order-service 服务之间的契约，还将验证 order-service 服务与该服务使用的其他服务之间的契约。

```
@RunWith(SpringRunner.class)
@SpringBootTest(properties = {
    "eureka.client.enabled: false"
})
@AutoConfigureStubRunner(ids = {
        "pl.piomin.services:account-service:+:stubs:8091",
        "pl.piomin.services:customer-service:+:stubs:8092",
        "pl.piomin.services:product-service:+:stubs:8093"
}, workOffline = true)
public class OrderScenarioBase {

    @Autowired
    private WebApplicationContext context;
    @MockBean
```

```
    private OrderRepository repository;
    @Before
    public void setup() {
        RestAssuredMockMvc.webAppContextSetup(context);
when(repository.countByCustomerId(Matchers.anyString())).
thenReturn(0);
        when(repository.save(Mockito.any(Order.class))).thenAnswer(new
Answer<Order>() {
            @Override
            public Order answer(InvocationOnMock invocation) throws
Throwable {
                Order o = invocation.getArgumentAt(0, Order.class);
                o.setId("12345");
                return o;
            }
        });
    }
}
```

## 13.7 性能测试

现在还有最后一种类型的自动化测试要讨论，也就是本章开头已经提到过的性能测试。有一些非常有趣的工具和框架可以帮助开发人员创建和运行此类测试。

性能测试工具有很多选择，特别是对于 HTTP API 测试来说尤其如此。我们无法讨论所有这些工具，但介绍一个框架也是很有用的。我们接下来将要讨论的框架就是 Gatling。

Gatling 是一个用 Scala 编写的开源性能测试工具。它允许开发人员以易于阅读和写入的领域特定语言（Domain-Specific Language，DSL）开发测试。它通过生成全面的图形负载报告从竞争者中脱颖而出，其报告说明了在测试用例期间收集到的所有指标。可以通过一些插件将 Gatling 与 Gradle、Maven 和 Jenkins 集成在一起。

### 1. 启用 Gatling

要为项目启用 Gatling 框架，应该在依赖项中包含 io.gatling.highcharts:gatling-charts-highcharts 工件。

### 2. 定义测试方案

每个 Gatling 测试套件都应该扩展 Simulation 类。在每个测试类中，可以使用 Gatling

Scala DSL 声明一个方案列表。我们通常声明可以调用 HTTP 端点的并发线程数，以及每个线程发送的整体请求数。在 Gatling 术语中，线程数由使用 atOnceUsers 方法设置的用户数决定。测试类应放在 src/test/scala 目录中。

假设我们要测试运行 20 个客户端的 order-service 服务公开的两个端点，其中每个端点按顺序发送 500 个请求，这意味着总共会发送 20×2×500=20000 个请求。通过在短时间内发送所有内容，开发人员就能够测试应用程序的性能。

以下测试方案是用 Scala 编写的，现在我们来仔细研究一下。在运行此测试之前，我们通过调用 HTTP API 创建了一些账户和产品，并通过 account-service 服务和 product-service 服务公开。由于它们连接到外部数据库，因而会自动生成 ID。为了提供一些测试数据，笔者已将它们复制到测试类中。具有账户和产品 ID 的列表都作为订阅源传递给测试方案。然后，在每次迭代期间，从列表中随机选择所需的值。我们的测试方案名为 AddAndConfirmOrder。它由两个 exec 方法组成。第一个是通过调用 POST /order HTTP 方法创建一个新订单。订单的 ID 由服务自动生成，因而应将其保存为属性。然后它可以在下一个 exec 方法中使用，该方法将通过调用 PUT /order/{id}端点来确认订单。在此测试之后唯一验证的是 HTTP 状态。

```
class OrderApiGatlingSimulationTest extends Simulation {

    val rCustomer = Iterator.continually(Map("customer" ->
List("5aa8f5deb44f3f188896f56f", "5aa8f5ecb44f3f188896f570",
"5aa8f5fbb44f3f188896f571",
"5aa8f620b44f3f188896f572").lift(Random.nextInt(4)).get))
    val rProduct = Iterator.continually(Map("product" ->
List("5aa8fad2b44f3f18f8856ac9","5aa8fad8b44f3f18f8856aca",
"5aa8fadeb44f3f18f8856acb","5aa8fae3b44f3f18f8856acc",
"5aa8fae7b44f3f18f8856acd","5aa8faedb44f3f18f8856ace",
"5aa8faf2b44f3f18f8856acf").lift(Random.nextInt(7)).get))

    val scn =
scenario("AddAndConfirmOrder").feed(rCustomer).feed(rProduct).
repeat(500, "n") {
        exec(
            http("AddOrder-API")
                .post("http://localhost:8090/order")
                .header("Content-Type", "application/json")
.body(StringBody("""{"productIds":["${product}"],"customerId":"${custo
mer}","status":"NEW"}"""))
```

```
                .check(status.is(200), jsonPath("$.id").saveAs("orderId"))
        )
        .
        exec(
            http("ConfirmOrder-API")
                .put("http://localhost:8090/order/${orderId}")
                .header("Content-Type", "application/json")
                .check(status.is(200))
        )
    }

    setUp(scn.inject(atOnceUsers(20))).maxDuration(FiniteDuration.
apply(10, "minutes"))

}
```

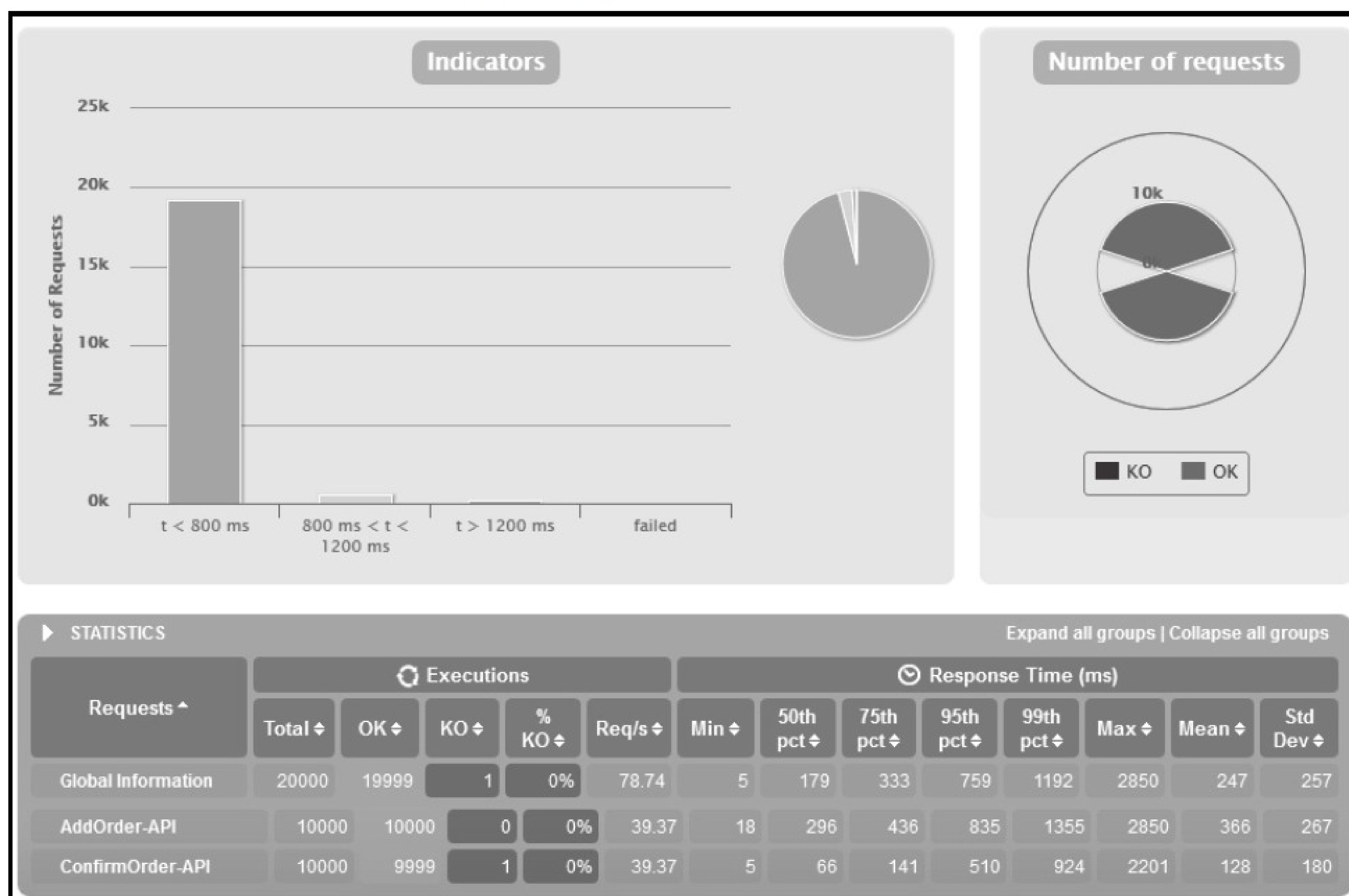
### 3．运行测试方案

在开发人员的计算机上运行 Gatling 性能测试有若干种不同的方法。其中之一是通过 Gradle 插件（Gradle 插件可以为在项目构建期间运行测试提供支持），也可以使用 Maven 插件或尝试从集成开发环境运行它。如果使用 Gradle 构建项目，还可以通过启动 io.gatling.app.Gatling 主类来定义仅运行测试的简单任务。以下是 gradle.build 文件中此类任务的定义。

```
task loadTest(type: JavaExec) {
    dependsOn testClasses
    description = "Load Test With Gatling"
    group = "Load Test"
    classpath = sourceSets.test.runtimeClasspath
    jvmArgs = [
        "-Dgatling.core.directory.binaries=
${sourceSets.test.output.classesDir.toStri ng()}"
    ]
    main = "io.gatling.app.Gatling"
    args = [
        "--simulation",
"pl.piomin.services.gatling.OrderApiGatlingSimulationTest",
        "--results-folder", "${buildDir}/gatling-results",
        "--binaries-folder", sourceSets.test.output.classesDir.toString(),
        "--bodies-folder",
sourceSets.test.resources.srcDirs.toList().first().toString() +
```

```
"/gatling/bodies",
    ]
}
```

现在，只需调用 gradle loadTest 命令即可运行该任务。当然，在运行这些测试之前，还需要启动所有示例微服务、MongoDB 和 discovery-service 服务。默认情况下，Gatling 将打印所有发送的请求、接收到的响应和最终的测试结果，包括时间统计信息以及 API 调用成功或失败的次数。如果需要更详细的信息，则应参考测试后生成的文件，这些文件位于 build/gatling-results 目录下。开发人员可以发现，其 HTML 文件会以图表和图形的形式提供可视化结果。首先显示的是一个汇总（如图 13.7 所示），其中包含生成的请求总数和按百分位数细分的最大响应时间。例如，在图 13.7 中可以看到，AddOrder API 的 95%响应中的最大响应时间为 835 毫秒。




| Requests | Executions | | | | | Response Time (ms) | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Total | OK | KO | % KO | Req/s | Min | 50th pct | 75th pct | 95th pct | 99th pct | Max | Mean | Std Dev |
| Global Information | 20000 | 19999 | 1 | 0% | 78.74 | 5 | 179 | 333 | 759 | 1192 | 2850 | 247 | 257 |
| AddOrder-API | 10000 | 10000 | 0 | 0% | 39.37 | 18 | 296 | 436 | 835 | 1355 | 2850 | 366 | 267 |
| ConfirmOrder-API | 10000 | 9999 | 1 | 0% | 39.37 | 5 | 66 | 141 | 510 | 924 | 2201 | 128 | 180 |

图 13.7　Gatling 的可视化结果

还有一些其他的有趣统计数据可视化。如图 13.8 所示的是值得我们特别关注的两个报告。其中第一个报告的图表显示了按平均响应时间分组的请求百分比，而第二个报告

则以百分位数显示了平均响应时间的时间线。

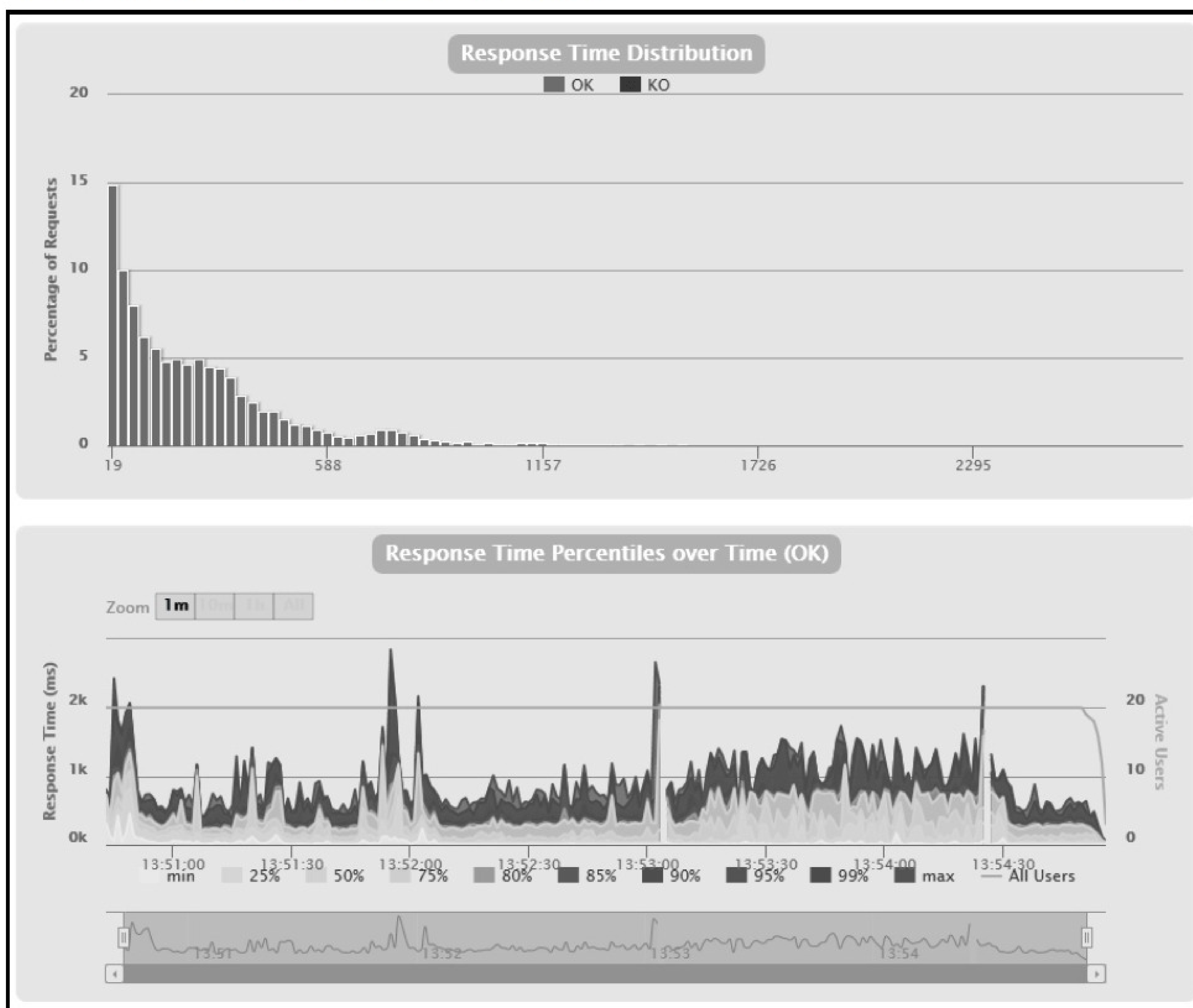



图 13.8　有趣的统计数据可视化结果

## 13.8　小　　结

本章介绍了一些可以帮助开发人员有效测试用 Java 编写的基于 REST 的应用程序的框架。这些解决方案中的每一个都已分配给特定类型的测试。我们专注于讨论与微服务严格相关的测试，如契约测试和组件测试。本章的主要目标是比较用于契约测试的两个最流行的框架，即 Pact 和 Spring Cloud Contract。尽管外表相似，但它们之间存在一些显著差异。本章采用了与前几章相同的示例应用程序，展示了它们之间最重要的相似点和

不同点。

微服务与自动化严格相关。开发人员应该意识到，从一体化应用程序到微服务的迁移使我们有机会重构代码，并且还可以提高自动化测试的质量和代码覆盖率。诸如 Mockito、Spring Test、Spring Cloud Contract 和 Pact 等框架在一起使用时，可以为开发人员提供一个非常强大的解决方案，开发基于 REST 的 Java 微服务的测试。自动化测试是持续集成/持续交付（CI/CD）过程的重要组成部分，第 14 章将详细讨论该过程。

# 第三部分

# Docker 支持和 Spring Cloud 平台

# 第 14 章　Docker 支持

在本书第一部分（第 1 章～第 3 章）中已经讨论了微服务架构和 Spring Cloud 项目的基础知识；在第二部分（第 4 章～第 13 章）中则研究了微服务架构中最常见的元素，并演示了如何使用 Spring Cloud 实现它们。到目前为止，我们已经详细介绍了与微服务迁移相关的一些重要主题，如集中式日志记录、分布式跟踪、安全性和自动化测试等。在掌握了这些知识之后，现在我们可以进入本书的最后一部分内容，讨论微服务作为云原生态开发方法的真正威力。例如，能够使用容器化工具将应用程序彼此隔离，在软件交付过程中实现持续部署（Continuous Deployment）以及轻松扩展应用程序的能力，所有这些优点都有助于微服务的快速普及。

在前面的章节中，我们使用过 Docker 镜像（Image）在本地计算机上运行第三方工具和解决方案。以此为切入点，本章将介绍 Docker 的主要概念，如它的基本命令和用例。此信息将有助于开发人员运行前面章节中提供的示例。然后，我们将讨论如何使用 Spring Boot 应用程序示例构建镜像，以及如何在本地计算机上的容器内运行它们。我们将使用简单的 Docker 命令以及更高级的工具，如 Jenkins 服务器（它可以帮助开发人员执行完整、持续的交付），并在组织中启用持续集成（Continuous Integration，CI）过程。最后，我们将介绍用于部署、扩展和管理容器化应用程序自动化的最流行的工具之一：Kubernetes。

我们的所有示例都将通过 Minikube 在单节点 Kubernetes 集群上以本地方式运行。

本章将要讨论的主题包括：

- 最实用的 Docker 命令。
- 构建包含 Spring Boot 微服务的 Docker 容器。
- 在 Docker 上运行 Spring Cloud 组件。
- 使用 Jenkins 和 Docker 进行持续集成/持续交付。
- 在 Minikube 上部署和运行微服务。

## 14.1　关于 Docker

Docker 是一个工具，可以帮助开发人员使用容器创建、部署和运行应用程序。它的设计旨在使开发人员和系统管理员按照 DevOps 理念受益。Docker 通过解决与之相关的

一些重要问题，帮助改进软件交付流程。其中一个问题是不可变交付（Immutable Delivery）的想法，这与人们认为软件要“对我有用”的认识有关。特别重要的是，开发人员在 Docker 中进行测试所使用的镜像与在生产模式中所使用的镜像相同，应该看到的唯一区别是在配置期间。对于基于微服务的系统，以不可变交付模式进行的软件交付似乎特别重要，因为有许多独立部署的应用程序。由于 Docker 的存在，开发人员现在可以专注于编写代码而无须担心目标操作系统（应用程序启动的地方）。因此，该操作可以使用相同的接口来部署、启动和维护所有应用程序。

Docker 越来越受欢迎还有很多其他原因。毕竟，容器化理念在信息技术领域并不是什么新鲜事。Linux 容器是在很多年前推出的，自 2008 年以来一直是其内核的一部分。但是，Docker 已经引入了其他技术所没有的若干新内容和解决方案。

首先，它提供了一个简单的接口，允许开发人员轻松将应用程序及其依赖项打包到单个容器中，然后跨 Linux 内核的不同版本和实现运行它。容器可以在任何支持 Docker 的服务器上以本地方式或远程运行，每个容器都可以在几秒钟内启动。开发人员也可以轻松地在其上运行每个命令而无须进入容器内部。此外，Docker 镜像的共享和分发机制允许开发人员提交他们的更改，并以与共享源代码相同的方式推送和提取镜像，如使用 Git。目前，几乎所有最流行的软件工具都作为镜像在 Docker Hub 上发布，其中有一些我们已成功用于运行示例应用程序所需的工具。

Docker 架构由一些基本定义和元素组成，最重要的就是容器（Container）。容器在单个计算机上运行，并与该计算机共享操作系统内核。它们包含在机器代码上运行特定软件所需的一切：运行时（Runtime）、系统工具、系统库和设置。容器是根据 Docker 镜像中的指令创建的。镜像就像是一种配方或模板，它定义了在容器上安装和运行必要软件的步骤。容器也可以与虚拟机进行比较，因为它们具有类似的资源隔离和分配优势。但是，它们虚拟化的是操作系统而不是硬件，这使得它们比虚拟机更具可移植性和效率。图 14.1 说明了 Docker 容器和虚拟机之间的架构差异。

所有容器都在称为 Docker 主机（Docker Host）的物理或虚拟机器上启动。反过来，Docker 主机运行一个 Docker 守护程序（Docker Daemon），它将监听 Docker 客户端通过 Docker API 发送的命令。Docker 客户端可能是命令行工具或其他软件，如 Kinematic。除了运行守护进程外，Docker 主机还负责存储缓存镜像和从这些镜像创建的容器。每个镜像都是从一组图层构建的。每个图层仅包含与父图层的增量差异。这样的镜像并不小，所以需要存储在别处，这个地方叫作 Docker 注册表（Docker Registry）。开发人员可以创建自己的私有存储库，也可以使用 Web 上提供的现有公共存储库。最受欢迎的存储库是 Docker Hub，它包含几乎所有必需的镜像。

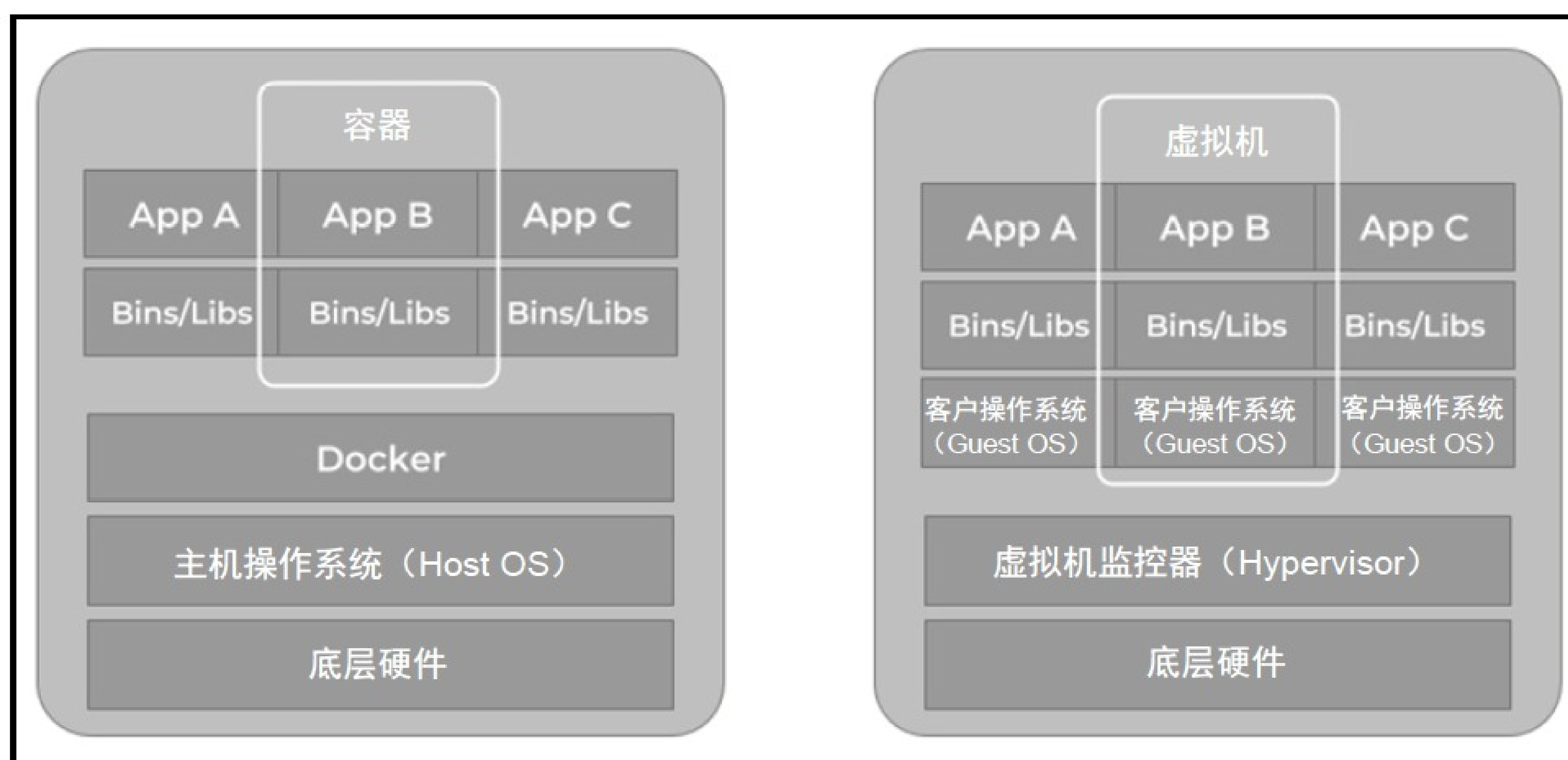



图 14.1　Docker 容器和虚拟机之间的架构差异

## 14.2　安装 Docker

适用于 Linux 的 Docker 安装说明和每个发行版本（https://docs.docker.com/install/#supported-platforms）有关。但是，有时开发人员必须在安装后运行 Docker 守护程序，可以通过调用以下命令来执行此操作。

```
dockerd --host=unix:///var/run/docker.sock --host=tcp://0.0.0.0:2375
```

本节将重点介绍 Windows 平台的说明。一般来说，在 Windows 或 Mac 上安装 Docker 社区版（Community Edition，CE）时，有两个可用选项。最快最简单的方法是使用 Docker for Windows，它可以在 https://www.docker.com/docker-windows 上找到，这是一个原生 Windows 应用程序，它为构建、交付和运行容器化应用程序提供了易于使用的开发环境。这绝对是最佳选择，因为它使用 Windows 原生的 Hyper-V 虚拟化和网络。但是，它也有一个缺点，那就是它仅适用于 Microsoft Windows 10 Professional 或 Enterprise 64 位。早期版本的 Windows 应使用 Docker Toolbox，开发人员可以访问 https://docs.docker.com/toolbox/toolbox_install_windows/地址并下载。该工具箱包括 Docker 平台、带 Docker Machine 的命令行、Docker Compose、Kitematic 和 VirtualBox 等。请注意，开发人员无法使用 Docker Toolbox 在 Windows 上以原生方式运行 Docker Engine，因为它使用了特定于 Linux 的内核功能。相反，开发人员必须使用 Docker Machine 命令（docker-machine），

该命令将在本地计算机上创建 Linux 虚拟机并使用 Virtual Box 运行它。开发人员的计算机可以使用虚拟地址（默认情况下为 192.168.99.100）访问此虚拟机。之前讨论的所有示例都与该 IP 地址提供的 Docker 工具集成在一起。

## 14.3　常用的 Docker 命令

在 Windows 上安装 Docker Toolbox 后，开发人员应该运行 Docker Quickstart Terminal。它可以完成所需的一切，包括创建和启动 Docker Machine 以及提供命令行界面。如果输入没有任何参数的 Docker 命令，则现在应该能够看到包含说明的可用 Docker 客户端命令的完整列表。以下是我们将要看到的命令类型。

- ❑ 运行和停止容器
- ❑ 列出并删除容器
- ❑ 提取和推送镜像
- ❑ 构建镜像
- ❑ 创建网络

### 14.3.1　运行和停止容器

在安装 Docker 之后，运行的第一个 Docker 命令通常是 docker run。如前文所述，此命令是以前示例中最常用的命令之一。此命令执行两项操作：它从注册表中提取并下载镜像定义（以防它未在本地缓存），并启动容器。可以为此命令设置许多选项，要了解这些选项，可以运行 docker run --help 命令并进行查看。某些选项具有单字母快捷键，这通常是最常用的选项。例如，选项-d 可以在后台运行一个容器，而选项-i 则可以保持 stdin 打开，即使它没有附加。如果容器必须对外公开任何端口，则可以使用激活选项-p 和定义<port_outside_container>:<port_inside_container>。某些镜像需要其他配置，这些配置一般来说可以通过使用-e 选项覆盖的环境变量来完成。使用--name 选项可以为容器设置友好名称，这通常也很有用，它可以为在其上运行其他命令提供方便。

在以下 Docker 命令示例中，启动了包含 Postgres 数据库的容器，并且创建了带密码的数据库用户，然后将它公开在端口 55432 上。现在，该 Postgres 数据库的地址为 192.168.99.100:55432。

```
$ docker run -d --name pg -e POSTGRES_PASSWORD=123456 -e
POSTGRES_USER=piomin -e POSTGRES_DB=example -p 55432:5432 postgres
```

带 Postgres 数据库的容器可以持久保存数据。对于存储数据的容器来说，外部应用程序如果要访问其数据，推荐机制是通过卷（Volume）。可以使用-v 选项将卷传递给容器，其中的值可以由冒号（:）分隔的字段组成。第一个字段是卷的名称，而第二个字段则是路径，即该文件或目录在容器中安装的位置。下一个有趣的选项是使用-m 选项限制为容器分配的最大RAM 的能力。以下是创建新卷并将其装入已启动容器的命令。最大RAM 容量设置为 500MB。在停止使用激活的选项--rm 之后，容器会自动删除，如下所示。

```
$ docker volume create pgdata
$ docker run --rm -it -e -m 500M -v pgdata:/var/lib/postgresql/data -p
55432:5432 postgres
```

可以使用 docker stop 命令停止每一个正在运行的容器。我们已经为容器设置了一个名称，因而可以轻松地将其用作标签，如下所示。

```
$ docker stop pg
```

由于容器的整个状态被写入磁盘，因而开发人员可以使用与停止之前完全相同的数据集再次运行它，如下所示。

```
$ docker start pg
```

如果只想重新启动容器，则可以使用以下命令而不是停止/启动容器。

```
$ docker restart pg
```

## 14.3.2　列出并删除容器

如果已经启动某些容器，则可能需要考虑在 Docker 计算机上显示所有正在运行的容器列表，此时应该使用 docker ps 命令，如图 14.2 所示。此命令将显示有关容器的一些基本信息，如公开端口列表和源镜像的名称。此命令仅打印当前启动的容器。如果希望查看已停止或处于非活动状态的容器，则可以在 Docker 命令中使用选项-a。

```
$ docker ps
CONTAINER ID    IMAGE                  COMMAND                  CREATED        STATUS          PORTS                                              NAMES
3c5c17a50936    jenkins/jenkins:lts    "/sbin/tini -- /us..."   5 days ago     Up 2 seconds    0.0.0.0:50000->50000/tcp, 0.0.0.0:38080->8080/tcp  jenkins
d07eb4195292    postgres               "docker-entrypoint..."   5 months ago   Up 16 seconds   0.0.0.0:35432->5432/tcp                            postgres
```

图 14.2　使用 docker ps 命令

如果不再需要容器，则可以使用 docker rm 命令将其删除。有时需要删除正在运行的容器，默认情况下不允许这样做。要强制使用此选项，则应该在 Docker 命令上设置 -f 选项，如下所示。

```
$ docker rm -f pg
```

开发人员应该记住，docker ps 命令仅删除容器。创建它的镜像仍然在本地缓存。这样的镜像可能占用大量空间，范围从数兆字节到几百兆字节。可以使用 docker rmi 命令删除每个镜像，并将镜像 ID 或名称作为参数，如下所示。

```
$ docker rmi 875263695ab8
```

虽然我们还没有创建任何 Docker 镜像，但在镜像创建过程中生成大量不需要的或未命名的镜像并不罕见。这些镜像可以很容易识别，因为它们会使用名称<none>表示。在 Docker 术语中，这些被称为悬空镜像/无效镜像（Dangling Image），可以使用以下命令轻松删除。此外，还可以使用 docker images 命令显示所有当前缓存镜像的列表，如下所示。

```
$ docker rmi $(docker images -q -f dangling=true)
```

## 14.3.3　提取和推送镜像

我们已经讨论过 Docker Hub，它是目前网上最大、最受欢迎的 Docker 存储库。它可以在 https://hub.docker.com 上找到。默认情况下，Docker 客户端将尝试提取该存储库的所有镜像。有许多经过认证的官方镜像可用于常见软件，如 Redis、Java、Nginx 或 Mongo，但开发人员也可以找到其他人创建的数十万个镜像。如果使用命令 docker run，则会从存储库中提取镜像，以防它未在本地缓存。还可以运行以下命令 docker pull，它仅负责下载镜像。

```
$ docker pull postgres
```

上述命令将下载最新版本的镜像（使用最新标记的名称）。如果开发人员想使用旧版本的 Postgres Docker 镜像，则应附加具有特定版本号的标签。可用版本的完整列表通常发布在镜像的站点上，在以下示例中则没有区别。开发人员可以访问 https://hub.docker.com/r/library/postgres/tags/以获取可用标签的列表。

```
$ docker pull postgres:9.3
```

运行并验证镜像后，开发人员应该考虑远程保存它。最适合它的地方当然是 Docker Hub。但是，有时开发人员可能希望将镜像存储在备用存储（如私有存储库）中。在推送镜像之前，必须使用注册表用户名、镜像名称及其版本号对其进行标记。以下命令将从 Postgres 源镜像创建一个名为 piomin/postgres 和 1.0 版本标记的新镜像。

```
$ docker tag postgres piomin/postgres:1.0
```

现在，如果运行 docker images 命令，则将看到两个具有相同 ID 的镜像。第一个名称为 Postgres，具有最新的标签；第二个名称为 piomin/postgres，标签为 1.0。重要的是，

piomin 是笔者在 Docker Hub 上的用户名。所以，在继续任何其他操作之前，我们应该先在那里注册镜像。

之后，我们还应该使用 docker login 命令登录我们的 Docker 客户端。在这里，系统将提示输入用于注册的用户名、密码和电子邮件地址。最后，可以使用以下 docker push 命令推送带标记的镜像。

```
$ docker push piomin/postgres:1.0
```

现在剩下要做的就是使用 Web 浏览器登录开发人员自己的 Docker Hub 账户，检查推送的镜像是否已经出现。如果一切正常，将看到一个新的公共存储库，其中包含开发人员自己的镜像。图 14.3 显示了当前推送到 Docker Hub 账户的镜像。

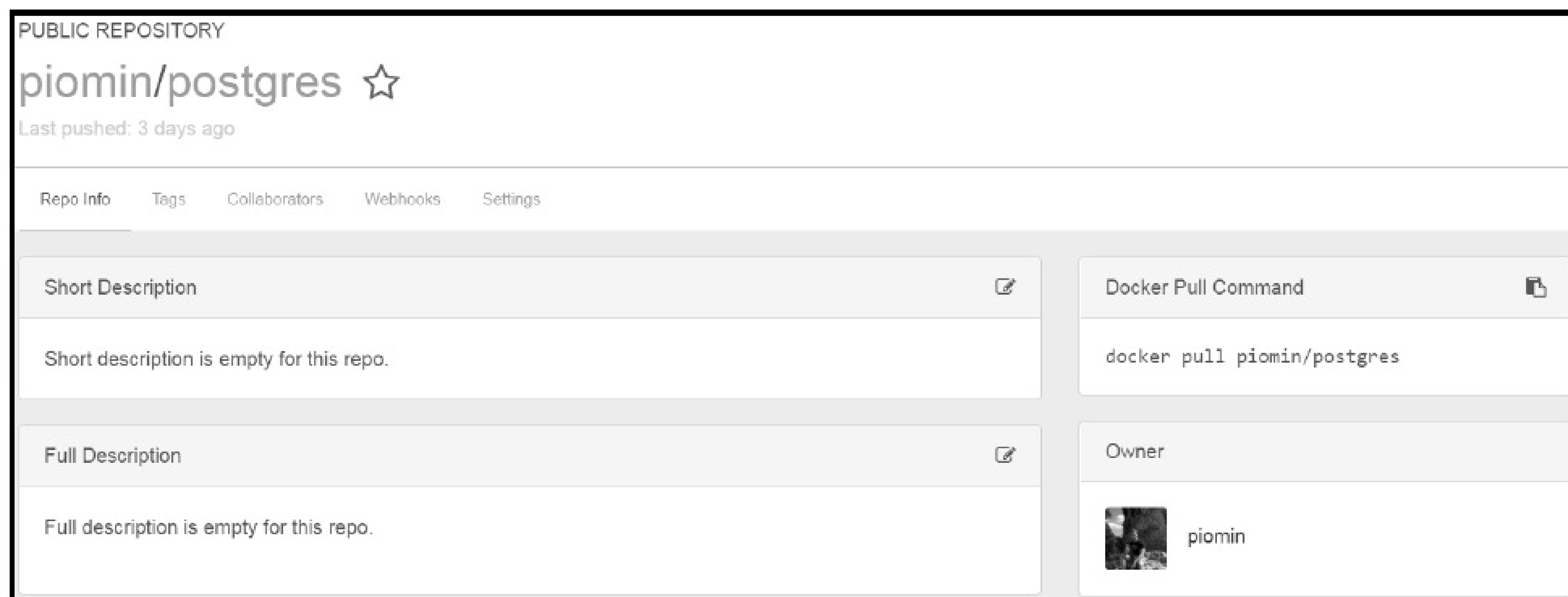



图 14.3　推送到 Docker Hub 账户的镜像

## 14.3.4　构建镜像

在 14.3.3 节中，我们将 Postgres Docker 镜像的副本推送到 Docker Hub 注册表。一般来说，我们推送的是从 Dockerfile 文件创建的自己的镜像，该文件定义了在容器上安装和配置软件时所需的所有指令。有关 Dockerfile 结构的细节将在后面讨论。目前重要的是用于构建 Docker 镜像的命令：docker build。此命令应在 Dockerfile 所在的同一目录中运行。构建新镜像时，建议使用-t 选项设置其名称和标记。以下命令将创建镜像 piomin/order-service，标记为 1.0 版本。该镜像可能会像上一个包含 Postgres 的镜像一样被推送到开发人员的 Docker Hub 账户，如下所示。

```
$ docker build -t piomin/order-service:1.0
```

### 14.3.5　创建网络

创建网络是 Docker 架构的一个重要方面，因为开发人员经常要在不同容器上运行的应用程序之间提供通信。常见用例是需要访问数据库的 Web 应用程序。我们现在将参考本书第 11 章“消息驱动的微服务”中已经介绍的另一个示例。它是 Apache Kafka 和 ZooKeeper 之间的通信。Kafka 需要 ZooKeeper，因为它将各种配置存储为 ZK 数据树中的键/值对，并在整个集群中使用它。如前文所述，开发人员首先必须创建一个自定义网络并在那里运行这两个容器。以下命令可用于在 Docker 主机上创建用户定义的网络。

```
$ docker network create kafka-network
```

在上述命令运行完毕之后，可以使用以下命令检查可用网络列表。默认情况下，Docker 会创建 3 个网络，因而开发人员应该看到 4 个网络，其名称分别为 bridge、host、none 和 kafka-network。

```
$ docker network ls
```

下一步是将网络名称传递给使用 docker run 命令创建的容器。它可以通过--network 参数实现，如下例所示。如果为两个不同的容器设置相同的网络名称，那么它们将在同一网络上启动。现在来分析一下这在实践中意味着什么。如果在一个容器中，那么开发人员可以通过其名称而不是使用其 IP 地址调用它，这就是为什么在使用 Apache Kafka 启动容器时可以将环境变量 ZOOKEEPER_IP 设置为 ZooKeeper。在此容器内启动的 Kafka 将连接默认端口上的 ZooKeeper 实例，如下所示。

```
$ docker run -d --name zookeeper --network kafka-net zookeeper:3.4
$ docker run -d --name kafka --network kafka-net -e
ZOOKEEPER_IP=zookeeper ches/kafka
```

## 14.4　创建具有微服务的 Docker 镜像

前文已经讨论了可用于运行、创建和管理容器的基本 Docker 命令。现在是时候创建和构建我们的第一个 Docker 镜像，以启动我们在第 13 章中介绍的示例微服务。为此，开发人员应该回到地址 https://github.com/piomin/sample-spring-cloud-comm.git 上的存储库，然后切换到 feign_with_discovery 分支（https://github.com/piomin/sample-spring-cloud-comm/tree/feign_with_discovery）。在该地址上，开发人员将找到每个微服务、网关和发现的 Dockerfile。

在讨论这些示例之前，开发人员应该参考 Dockerfile 的一些资料来理解其基本命令。实际上，Dockerfile 并不是构建 Docker 镜像的唯一方法，在第 14.4.3 节中还将演示如何使用 Maven 插件创建具有微服务的镜像。

## 14.4.1　Dockerfile

Docker 可以通过读取 Dockerfile 中提供的指令自动构建镜像，Dockerfile 是一个文档，其中包含在命令行上调用以组成镜像的所有命令。所有这些命令都必须以 Dockerfile 规范中定义的关键字开头。以下是最常用指令的列表，它们按照在 Dockerfile 中找到它们的顺序执行。在表 14.1 中，我们还可以附加一些注释，这些注释的后面必须跟着＃字符。

表 14.1　Dockerfile 指令

| 指　　令 | 说　　明 |
| --- | --- |
| FROM | 这将初始化一个新的构建阶段，并为后续指令设置基本镜像。实际上，每个有效的 Dockerfile 都必须以 FROM 指令开头 |
| MAINTAINER | 设置生成镜像的作者身份。该指令已经建议不再使用，因而仅可在一些较旧的镜像中发现它。开发人员应该使用 LABEL 指令而不是 MAINTAINER，如下所示：LABEL maintainer ="piotr.minkowski@gmail.com" |
| RUN | 执行 Linux 命令，在当前镜像之上的新层中配置和安装所需软件，然后提交结果。它可以有两种形式：RUN <command>或 RUN ["executable","param1","param2"] |
| ENTRYPOINT | 这将配置一个最终脚本，在引导容器时，它将作为可执行文件运行。它覆盖了使用 CMD 指定的所有元素，并有两种形式：ENTRYPOINT ["executable","param1","param2"]和 ENTRYPOINT 命令 param1 param2。值得注意的是，只有 Dockerfile 中的最后一个 ENTRYPOINT 指令才会产生影响 |
| CMD | Dockerfile 只能包含一条 CMD 指令。这条指令将使用 JSON 数组格式为 ENTRYPOINT 提供默认参数 |
| ENV | 这以键/值的形式设置容器的环境变量 |
| COPY | 将新文件或目录从给定源路径复制到由目标路径定义的路径中容器内的文件系统。它具有以下形式：COPY[--chown=<user>:<group>] <src>...<DEST> |
| ADD | 这是 COPY 指令的替代方案。它允许比 COPY 多一点功能，例如，它允许<src>成为 URL 地址 |
| WORKDIR | 这可以为 RUN、CMD、ENTRYPOINT、COPY 和 ADD 设置工作目录 |
| EXPOSE | 这负责通知 Docker 容器在运行时侦听指定的网络端口。它实际上并不发布端口。端口将通过 docker run 命令上的-p 选项发布 |
| VOLUME | 这将创建具有指定名称的安装点。卷是在 Docker 容器中持久保存数据的首选机制 |
| USER | 设置运行镜像及 RUN、CMD 和 ENTRYPOINT 指令时使用的用户名和用户组（可选） |

现在来看一看它在实践中是如何运作的。我们应该为每个微服务定义一个 Dockerfile，并将其放在 Git 项目的根目录中。以下是为 account-service 服务创建的 Dockerfile。

```
FROM openjdk:8u151-jdk-slim-stretch
MAINTAINER Piotr Minkowski <piotr.minkowski@gmail.com>
ENV SPRING PROFILES ACTIVE zone1
ENV EUREKA DEFAULT ZONE http://localhost:8761/eureka/
ADD target/account-service-1.0-SNAPSHOT.jar app.jar
ENTRYPOINT ["java", "-Xmx160m", "-jar",
"-Dspring.profiles.active=${SPRING PROFILES ACTIVE}",
"-Deureka.client.serviceUrl.defaultZone=${EUREKA DEFAULT ZONE}",
"/app.jar"]
EXPOSE 8091
```

前面的例子并不复杂。它只将由微服务生成的胖 JAR 文件添加到 Docker 容器中，并使用 java -jar 命令作为 ENTRYPOINT。为了更好地理解它，下文将逐步进行分析。我们的示例 Dockerfile 将执行以下指令。

- 该镜像扩展了现有的 OpenJDK 镜像，该镜像是 Java 平台标准版（Java Platform Standard Edition）的官方开源实现。OpenJDK 镜像有很多种，可用镜像变体之间的主要区别在于它们的大小。标记为 8u151-jdk-slim-stretch 的镜像提供了 JDK 8，并包含运行 Spring Boot 微服务所需的所有库。它也比 8u151-jdk 版本 Java 的基本镜像小得多。
- 在这里，我们使用 docker run 命令的-e 选项定义了两个可以在运行时覆盖的环境变量。第一个是活动的 Spring 配置文件名称，默认情况下它将使用 zone1 值初始化。第二个是发现服务器的地址，默认情况下等于 http://localhost:8761/eureka/。
- 胖 JAR 文件包含所有必需的依赖项以及应用程序的二进制文件。因此，我们必须使用 ADD 指令将已生成的 JAR 文件放入容器中。
- 我们将容器配置为可执行 Java 应用程序。定义的 ENTRYPOINT 相当于在本地计算机上运行以下命令。

```
java -Xmx160m -jar -Dspring.profiles.active = zone1 -
Deureka.client.serviceUrl.defaultZone = http: //localhost: 8761/eureka/
app.jar
```

- 使用 EXPOSE 指令，我们告知 Docker 它可能会公开应用程序的 HTTP API，该 API 在端口 8091 上的容器内可用。

### 14.4.2　运行容器化微服务

假设我们为每个服务准备了一个有效的 Dockerfile，下一步是使用 mvn clean install 命令构建整个 Maven 项目，然后为每个服务构建一个 Docker 镜像。

构建 Docker 镜像时，应始终位于每个微服务源代码的 root 目录中。在基于微服务的系统中运行的第一个容器必须是发现服务器。其 Docker 镜像已被命名为 piomin/discovery-service。在运行 Docker 的 build 命令之前，应转到模块 discovery-service。这个 Dockerfile 比其他微服务稍微简单一些，因为在容器内没有设置环境变量，如下所示。

```
FROM openjdk:8u151-jdk-slim-stretch
MAINTAINER Piotr Minkowski <piotr.minkowski@gmail.com>
ADD target/discovery-service-1.0-SNAPSHOT.jar app.jar
ENTRYPOINT ["java", "-Xmx144m", "-jar", "/app.jar"]
EXPOSE 8761
```

这里仅执行 5 个步骤，可以在运行 docker build 命令之后在目标镜像构建期间生成的日志中看到这些步骤。如果一切正常，开发人员应该看到在 Dockerfile 中定义的所有 5 个步骤的进度以及以下最终消息，这些消息告诉开发人员镜像已成功构建和标记。

```
$ docker build -t piomin/discovery-service:1.0 .
Sending build context to Docker daemon 39.9MB
Step 1/5 : FROM openjdk:8u151-jdk-slim-stretch
8u151-jdk-slim-stretch: Pulling from library/openjdk
8176e34d5d92: Pull complete
2208661344b7: Pull complete
99f28966f0b2: Pull complete
e991b55a8065: Pull complete
aee568884a84: Pull complete
18b6b371c215: Pull complete
Digest:
sha256:bd394fdc76e8aa73adba2a7547fcb6cde3281f70d6b3cae6fa62ef1fbde327e3
Status: Downloaded newer image for openjdk:8u151-jdk-slim-stretch
 ---> 52de5d98a41d
Step 2/5 : MAINTAINER Piotr Minkowski <piotr.minkowski@gmail.com>
 ---> Running in 78fc78cc21f0
 ---> 0eba7a369e43
Removing intermediate container 78fc78cc21f0
Step 3/5 : ADD target/discovery-service-1.0-SNAPSHOT.jar app.jar
 ---> 1c6a2e04c4dc
Removing intermediate container 98138425b5a0
```

```
Step 4/5 : ENTRYPOINT java -Xmx144m -jar /app.jar
 ---> Running in 7369ba693689
 ---> c246470366e4
Removing intermediate container 7369ba693689
Step 5/5 : EXPOSE 8761
 ---> Running in 74493ae54220
 ---> 06af6a3c2d41
Removing intermediate container 74493ae54220
Successfully built 06af6a3c2d41
Successfully tagged piomin/discovery-service:1.0
```

如果成功构建了一个镜像，则应该运行它。我们建议创建一个网络，其中将启动包含微服务的所有容器。要在新创建的网络中启动容器，必须使用--network 参数将其名称传递给 docker run 命令。要检查容器是否已成功启动，可以运行 docker logs 命令。此命令将应用程序记录的所有行打印到控制台，如下所示。

```
$ docker network create sample-spring-cloud-network
$ docker run -d --name discovery -p 8761:8761 --network sample-spring-
cloud-network piomin/discovery-service:1.0
de2fac673806e134faedee3c0addaa31f2bbadcffbdff42a53f8e4ee44ca0674
$ docker logs -f discovery
```

下一步是使用我们的 4 个微服务——account-service 服务、customer-service 服务、order-service 服务和 product-service 服务来构建和运行容器。每个服务的过程都是相同的。例如，如果想要构建 account-service 服务，则首先需要转到示例项目源代码的该目录中。这里的 build 命令与发现服务的命令相同，唯一的区别在于镜像名称，如下面的代码段所示。

```
$ docker build -t piomin/account-service:1.0
```

运行 Docker 镜像的命令对于 discovery-service 来说有点复杂。在这种情况下，必须将 Eureka 服务器的地址传递给起始容器。由于此容器与发现服务容器在同一网络中运行，因此，开发人员可以使用其名称而不是其 IP 地址或任何其他标识符。或者，也可以使用-m 参数设置容器的内存限制，如设置为 256MB。最后，还可以使用 docker logs 命令查看运行在容器上的应用程序生成的日志，如下所示。

```
$ docker run -d --name account -p 8091:8091 -e
EUREKA_DEFAULT_ZONE=http://discovery:8761/eureka -m 256M --network
sample-spring-cloud-network piomin/account-service:1.0
$ docker logs -f account
```

对于所有其他微服务，应该重复与前面描述相同的步骤。最终结果是 5 个正在运行

的容器，这可以使用 docker ps 命令显示，如图 14.4 所示。

```
$ docker ps
CONTAINER ID     IMAGE                             COMMAND                   CREATED            STATUS           PORTS                      NAMES
93a7423f9d8b     piomin/order-service:1.0          "java -Xmx160m -ja..."    12 minutes ago     Up 11 minutes    0.0.0.0:8090->8090/tcp     order
c313d0a426bc     piomin/product-service:1.0        "java -Xmx160m -ja..."    14 minutes ago     Up 13 minutes    0.0.0.0:8093->8093/tcp     product
dae7e9271c1d     piomin/customer-service:1.0       "java -Xmx160m -ja..."    16 minutes ago     Up 15 minutes    0.0.0.0:8092->8092/tcp     customer
8020c04e6bf0     piomin/account-service:1.0        "java -Xmx160m -ja..."    19 minutes ago     Up 18 minutes    0.0.0.0:8091->8091/tcp     account
de2fac673806     piomin/discovery-service:1.0      "java -jar /discov..."    38 minutes ago     Up 37 minutes    0.0.0.0:8761->8761/tcp     discovery
```

图 14.4　使用 docker ps 命令显示的 5 个正在运行的容器

所有微服务都在 Eureka 服务器中注册。Eureka 仪表板的地址为 http://192.168.99.100:8761/，如图 14.5 所示。

Instances currently registered with Eureka

| Application | AMIs | Availability Zones | Status |
|---|---|---|---|
| ACCOUNT-SERVICE | n/a (1) | (1) | UP (1) - 8020c04e6bf0:account-service:8091 |
| CUSTOMER-SERVICE | n/a (1) | (1) | UP (1) - dae7e9271c1d:customer-service:8092 |
| ORDER-SERVICE | n/a (1) | (1) | UP (1) - 93a7423f9d8b:order-service:8090 |
| PRODUCT-SERVICE | n/a (1) | (1) | UP (1) - c313d0a426bc:product-service:8093 |

图 14.5　在 Eureka 服务器中注册的服务实例

在这里有我们提到过的另一个有趣的 Docker 命令：docker stats。此命令打印一些与已启动容器相关的统计信息，如内存或 CPU 使用情况。如果使用该命令的--format 参数，则可以自定义打印统计信息的方式。例如，可以打印容器名称而不是其 ID。在运行该命令之前，可以执行一些测试，以检查一切是否正常工作。开发人员可以注意检查在容器上启动的微服务之间的通信是否已成功完成。此外，如果想要尝试调用 customer-service 服务的端点 GET /withAccounts/{id}（它将调用 account-service 服务公开的端点），则可以运行以下命令。

```
docker stats --format "table
{{.Name}}\t{{.Container}}\t{{.CPUPerc}}\t{{.MemUsage}}"
```

如图 14.6 所示的是 docker stats 命令的打印结果。

```
NAME          CONTAINER       CPU %     MEM USAGE / LIMIT
order         f2c679a1c866    0.14%     208.9MiB / 256MiB
product       83f1761de51f    0.22%     199.6MiB / 256MiB
customer      59e747bdd022    0.23%     210.2MiB / 256MiB
account       28cc8ffcc5f7    0.15%     209.8MiB / 256MiB
discovery     de2fac673806    1.31%     255.3MiB / 1.955GiB
```

图 14.6　docker stats 命令的打印结果

## 14.4.3　使用 Maven 插件构建镜像

如前文所述，Dockerfile 不是创建和构建容器的唯一方法，还有一些其他方法可用，

如使用 Maven 插件。我们有许多可用于构建镜像的插件，它们与 mvn 命令一起使用。 其中一个比较受欢迎的是 com.spotify:docker-Maven-plugin 插件。它在配置中具有等效标记，可用于代替 Dockerfile 指令。pom.xml 文件中用于 account-service 服务的插件配置如下。

```
<plugin>
 <groupId>com.spotify</groupId>
 <artifactId>docker-maven-plugin</artifactId>
 <version>1.0.0</version>
 <configuration>
 <imageName>piomin/${project.artifactId}</imageName>
 <imageTags>${project.version}</imageTags>
 <baseImage>openjdk:8u151-jdk-slim-stretch</baseImage>
 <entryPoint>["java", "-Xmx160m", "-jar",
"- Dspring.profiles.active=${SPRING PROFILES ACTIVE}",
"- Deureka.client.serviceUrl.defaultZone=${EUREKA DEFAULT ZONE}",
"/${project.build.finalName}.jar"] </entryPoint>
 <env>
  <SPRING_PROFILES_ACTIVE>zone1</SPRING_PROFILES_ACTIVE>
<EUREKA_DEFAULT_ZONE>http://localhost:8761/eureka/</EUREKA_DEFAULT_ZONE>
 </env>
 <exposes>8091</exposes>
 <maintainer>piotr.minkowski@gmail.com</maintainer>
 <dockerHost>https://192.168.99.100:2376</dockerHost>
<dockerCertPath>C:\Users\Piotr\.docker\machine\machines\default
</docker CertPath>
 <resources>
  <resource>
   <directory>${project.build.directory}</directory>
   <include>${project.build.finalName}.jar</include>
  </resource>
 </resources>
 </configuration>
</plugin>
```

可以在 Maven 的 build 命令期间调用此插件。如果想要在构建应用程序之后构建 Docker 镜像，可以使用以下 Maven 命令。

```
$ mvn clean install docker: build
```

或者，开发人员也可以设置 dockerDirectory 标记，以便基于 Dockerfile 执行构建。无论选择哪种方法，效果都是一样的。使用应用程序构建的任何新镜像都可以在 Docker 机器上使用。使用 docker-maven-plugin 时，可以通过将 pushImage 设置为 true 来强制自动

将镜像推送到存储库，如下所示。

```
<plugin>
 <groupId>com.spotify</groupId>
 <artifactId>docker-maven-plugin</artifactId>
 <version>1.0.0</version>
 <configuration>
  <imageName>piomin/${project.artifactId}</imageName>
  <imageTags>${project.version}</imageTags>
  <pushImage>true</pushImage>
  <dockerDirectory>src/main/docker</dockerDirectory>
  <dockerHost>https://192.168.99.100:2376</dockerHost>
<dockerCertPath>C:\Users\Piotr\.docker\machine\machines\default
</docker CertPath>
  <resources>
   <resource>
    <directory>${project.build.directory}</directory>
    <include>${project.build.finalName}.jar</include>
   </resource>
  </resources>
 </configuration>
</plugin>
```

### 14.4.4　高级 Docker 镜像

到目前为止，我们已经构建了相当简单的 Docker 镜像。但是，有时需要创建更高级的镜像，我们需要这样的镜像用于持续交付（Continuous Delivery）演示。这个 Docker 镜像将作为 Jenkins 从属（Slave）服务器运行，并将连接到 Jenkins 主（Master）服务器，作为 Docker 容器启动。我们还没有在 Docker Hub 上找到这样的镜像，所以我们自己创建了一个。在这里，镜像必须包含 Git、Maven、JDK8 和 Docker。这些是使用 Jenkins 从属服务器构建示例微服务所需的所有工具。我们将在本章的后面部分简要介绍使用 Jenkins 服务器进行持续交付的基本知识。目前，我们将专注于构建所需的镜像。以下是 Dockerfile 中提供的镜像的完整定义。

```
FROM docker:18-dind
MAINTAINER Piotr Minkowski <piotr.minkowski@gmail.com>
ENV JENKINS_MASTER http://localhost:8080
ENV JENKINS_SLAVE_NAME dind-node
ENV JENKINS_SLAVE_SECRET ""
ENV JENKINS_HOME /home/jenkins
```

```
ENV JENKINS_REMOTING_VERSION 3.17
ENV DOCKER_HOST tcp://0.0.0.0:2375

RUN apk --update add curl tar git bash openjdk8 sudo

ARG MAVEN_VERSION=3.5.2
ARG USER_HOME_DIR="/root"
ARG
SHA=707b1f6e390a65bde4af4cdaf2a24d45fc19a6ded00fff02e91626e3e42ceaff
ARG
BASE URL=https://apache.osuosl.org/maven/maven-3/${MAVEN VERSION}/binaries
RUN mkdir -p /usr/share/maven /usr/share/maven/ref \
 && curl -fsSL -o /tmp/apache-maven.tar.gz ${BASE URL}/apache-maven-
${MAVEN VERSION}-bin.tar.gz \
 && echo "${SHA} /tmp/apache-maven.tar.gz" | sha256sum -c - \
 && tar -xzf /tmp/apache-maven.tar.gz -C /usr/share/maven --strip-
components=1 \
 && rm -f /tmp/apache-maven.tar.gz \
 && ln -s /usr/share/maven/bin/mvn /usr/bin/mvn
ENV MAVEN HOME /usr/share/maven
ENV MAVEN CONFIG "$USER HOME DIR/.m2"

RUN adduser -D -h $JENKINS HOME -s /bin/sh jenkins jenkins && chmod
a+rwx $JENKINS HOME
RUN echo "jenkins ALL=(ALL) NOPASSWD: /usr/local/bin/dockerd" >
/etc/sudoers.d/00jenkins && chmod 440 /etc/sudoers.d/00jenkins
RUN echo "jenkins ALL=(ALL) NOPASSWD: /usr/local/bin/docker" >
/etc/sudoers.d/01jenkins && chmod 440 /etc/sudoers.d/01jenkins
RUN curl --create-dirs -sSLo /usr/share/jenkins/slave.jar
http://repo.jenkins-ci.org/public/org/jenkins-ci/main/remoting/$JENKINS
_REMOTING_VERSION/remoting-$JENKINS_REMOTING_VERSION.jar && chmod 755
/usr/share/jenkins && chmod 644 /usr/share/jenkins/slave.jar

COPY entrypoint.sh /usr/local/bin/entrypoint
VOLUME $JENKINS_HOME
WORKDIR $JENKINS_HOME
USER jenkins
ENTRYPOINT ["/usr/local/bin/entrypoint"]
```

现在我们来分析一下发生了什么。在这里，我们扩展了 Docker 基础镜像。这是一个非常聪明的解决方案，因为该镜像现在已经在 Docker 中提供了 Docker。虽然通常不建议在 Docker 中运行 Docker，但是也有一些理想的用例，例如，使用 Docker 进行持续交付。

除了 Docker 之外，还使用 RUN 指令在镜像上安装其他软件，如 Git、JDK、Maven 或 Curl。我们还添加了一个操作系统用户，它在 dockerd 脚本中具有 sudoers 权限，该脚本负责在机器上运行 Docker 守护程序。这不是必须在正在运行的容器中启动的唯一进程，还需要使用 Jenkins 从属服务器启动 JAR。这两个命令在 entrypoint.sh 中执行，它被设置为镜像的 ENTRYPOINT。有关此 Docker 镜像的完整源代码，请访问 GitHub，其网址为 https://github.com/piomin/jenkins-slave-dind-jnlp.git。开发人员不必从源代码构建它，只需使用以下命令从笔者的 Docker Hub 账户下载已准备好的镜像。

```
docker pull piomin / jenkins-slave-dind-jnlp
```

以下是 Docker 镜像中的脚本 entrypoint.sh，它启动了 Docker 守护进程和 Jenkins 从属服务器。

```
#!/bin/sh
set -e
echo "starting dockerd..."
sudo dockerd --host=unix:///var/run/docker.sock --
host=tcp://0.0.0.0:2375 --storage-driver=vfs &
echo "starting jnlp slave..."
exec java -jar /usr/share/jenkins/slave.jar \
 -jnlpUrl $JENKINS_URL/computer/$JENKINS_SLAVE_NAME/slave-agent.jnlp \
 -secret $JENKINS_SLAVE_SECRET
```

# 14.5　持续交付

迁移到基于微服务的架构的关键优势之一是能够快速交付软件。这应该是在组织中实现持续交付或持续部署过程的主要动机。简而言之，持续交付流程是一种尝试自动化软件交付的所有阶段（如构建、测试代码和发布应用程序）的方法。有许多工具可以促进这一过程。其中一个是 Jenkins——一个用 Java 编写的开源自动化服务器。Docker 可以将持续集成（Continuous Integration，CI）或持续交付（Continuous Delivery，CD）流程提升到更高的水平。例如，不可变交付就是 Docker 最重要的优势之一。

## 14.5.1　将 Jenkins 与 Docker 集成

这里的主要目标是使用 Jenkins 和 Docker 在本地设计和运行持续交付流程。在此过程中有 4 个元素。第一个是微服务的源代码存储库，这已经准备好了，可以在 GitHub 上

找到。第二个元素是 Jenkins，它需要运行和配置。Jenkins 是持续交付系统的关键要素。它必须从 GitHub 存储库下载应用程序的源代码，构建它，然后将生成的 JAR 文件放在 Docker 镜像中，并将该镜像推送到 Docker Hub，最后运行包含微服务的容器。此过程中的所有任务都直接在 Jenkins 主服务器上而不是在其从属节点上执行。Jenkins 及其从属节点都是作为 Docker 容器启动的。该解决方案的架构说明如图 14.7 所示。

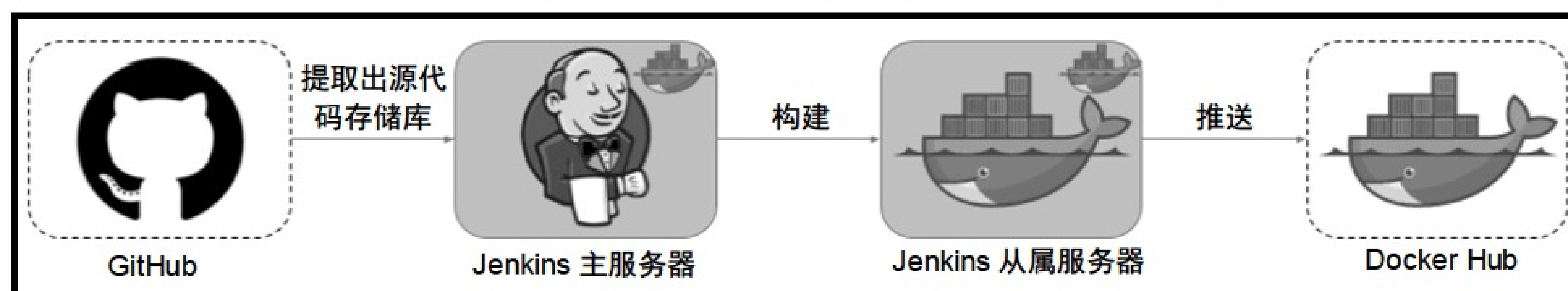



图 14.7　将 Jenkins 与 Docker 集成的解决方案架构

值得一提的是，Jenkins 是建立在插件概念基础上的，其核心是一个简单的自动构建引擎。Jenkins 的真正威力在于其插件，在其更新中心中有数百个插件。目前，由于我们要使用的是 Jenkins 服务器，因而我们将仅讨论一些对此有用的插件。开发人员需要安装以下插件才能在 Docker 容器中构建和运行微服务。

- 管道（Pipeline）：这是一套插件，它允许开发人员按照管道即代码（Pipeline as Code）的概念，使用 Groovy 脚本创建自动化（https://wiki.jenkins.io/display/JENKINS/Pipeline+Plugin）。
- Docker 管道（Docker Pipeline）：它允许开发人员在管道中构建 Docker 容器（https://wiki.jenkins.io/display/JENKINS/Docker+Pipeline+Plugin）。
- Git：它可以将 Git 与 Jenkins 集成（https://wiki.jenkins.io/display/JENKINS/Git+Plugin）。
- Maven 集成（Maven Integration）：它在使用 Maven 和 Jenkins 构建应用程序时可以提供一些有用的命令（https://plugins.jenkins.io/maven-plugin）。

可以通过用户界面仪表板配置所需的插件。配置可以在启动后或通过 Manage Jenkins | Manage Plugins（管理 Jenkins | 管理插件）进行。要在本地运行 Jenkins，需要使用其 Docker 镜像。以下命令创建一个名为 jenkins 的网络并启动 Jenkins 主容器，在端口 38080 上公开用户界面仪表板。请注意，在第一次启动 Jenkins 容器并使用其 Web 控制台时，需要使用初始生成的密码来设置它。开发人员可以通过调用 docker logs jenkins 命令从 Jenkins 日志中轻松检索此密码，如下所示。

```
$ docker network create jenkins
$ docker run -d --name jenkins -p 38080:8080 -p 50000:50000 --network
jenkins jenkins/jenkins:lts
```

一旦成功配置了 Jenkins 主服务器及其所需的插件，开发人员就需要添加新的从属节点。要执行此操作，可以转到 Manage Jenkins | Manage Nodes（管理 Jenkins | 管理节点）部分，然后选择 New Node（新建节点）。在显示的表单中，必须设置 /home/jenkins 作为远程根目录，并选择 launch agent via Java Web Start（通过 Java Web Start 启动代理）作为启动方法。现在可以启动带 Jenkins 从属节点的 Docker 容器，如前文所述。请注意，必须覆盖两个指示从属名称和密钥的环境变量。name 参数将在节点创建期间设置，而密钥则由服务器自动生成。开发人员可以查看节点的详细信息页面以获取更多信息，如图 14.8 所示。

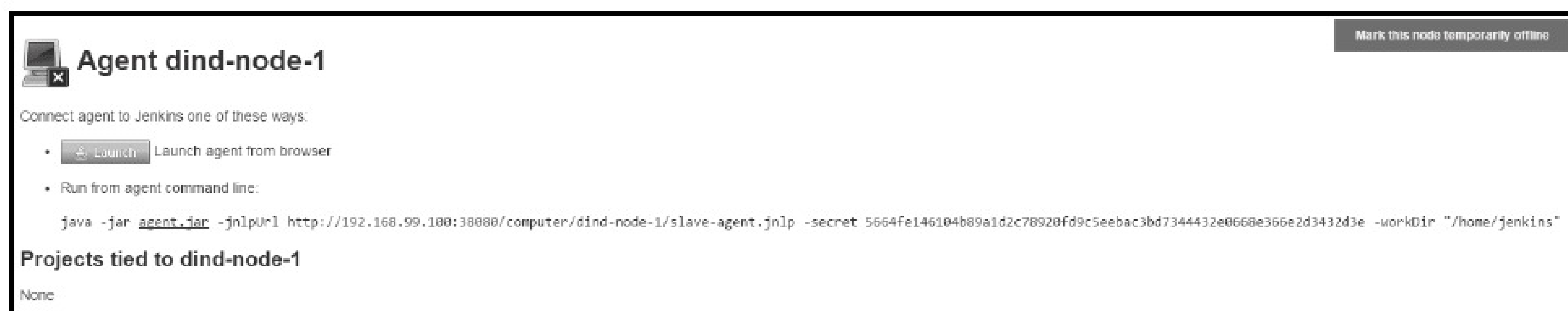



图 14.8　查看节点的详细信息页面

以下是 Docker 命令，它将在 Docker 中使用 Docker 启动带 Jenkins 从属节点的容器。

```
$ docker run --privileged -d --name slave --network jenkins -e
JENKINS_SLAVE_SECRET=5664fe146104b89a1d2c78920fd9c5eebac3bd7344432e0668
e366e2d3432d3e -e JENKINS_SLAVE_NAME=dind-node-1 -e
JENKINS_URL=http://jenkins:38080 piomin/jenkins-slave-dind-jnlp
```

上述对 Jenkins 配置的简短介绍应该可以帮助开发人员在自己的机器上重复之前讨论的持续交付流程。请记住，我们只查看了与 Jenkins 相关的一些方面（包括设置），这些方面允许开发人员为自己的基于微服务的系统设置 CI 或 CD 环境。如果开发人员有兴趣更深入地探讨此主题，可以访问 https://jenkins.io/doc 以参考其中的资料。

## 14.5.2　构建管道

在旧版本的 Jenkins 服务器中，基本工作单元是作业（Job）。目前，其主要功能是将管道定义为代码。此更改与 IT 架构中更现代的趋势有关，这些趋势认为，应用程序交付与正在交付的应用程序一样重要。由于应用程序堆栈的所有组件都已自动化并在版本控制系统中表示为代码，因而可以为 CI 或 CD 管道利用相同的优势。

Jenkins Pipeline 提供了一组工具，用于将简单和更高级的交付管道建模为代码。这种管道的定义通常写入一个名为 Jenkinsfile 的文本文件中。它支持领域特定语言（DSL），

并通过共享库（Shared Libraries）功能提供其他特定步骤。Pipeline 支持两种语法：声明式语法（Declarative）和脚本管道（Scripted Pipeline）。其中，声明式语法是在 Pipeline 2.5 中引入的。无论使用哪种语法，它都将在逻辑上划分为阶段（Stage）和步骤（Step）。步骤是管道中最基本的部分，因为它们告诉 Jenkins 要做什么。阶段按逻辑组合了若干个步骤，然后显示在管道的结果屏幕上。以下代码是脚本管道的示例，并定义了 account-service 服务的构建过程。

必须为其他微服务创建类似的定义。所有这些定义都位于每个应用程序源代码（如 Jenkinsfile）的 root 目录中。

```
node('dind-node-1') {
 withMaven(maven:'M3') {
  stage('Checkout') {
   git url: 'https://github.com/piomin/sample-spring-cloud-comm.git',
credentialsId: 'github-piomin',branch: 'master'
  }

  stage('Build') {
   dir('account-service') {
    sh 'mvn clean install'
   }
   def pom = readMavenPom file:'pom.xml'
   print pom.version
   env.version = pom.version
   currentBuild.description = "Release: ${env.version}"
  }

  stage('Image') {
   dir ('account-service') {
    def app = docker.build "piomin/account-service:${env.version}"
    app.push()
   }
  }

  stage ('Run') {
   docker.image("piomin/account-service:${env.version}").run('-p  8091:
8091 -d --name account --network sample-spring-cloud-network')
  }

 }
}
```

上述示例代码中的定义分为 4 个阶段。第一个阶段是 Checkout，在该阶段中克隆了包含所有示例应用程序源代码的 Git 存储库。第二阶段是 Build，在该阶段中从 account-service 服务模块构建了一个应用程序，然后从 root 的 pom.xml 文件中读取了整个 Maven 项目的版本号。第三个阶段是 Image 阶段，在该阶段中从 Dockerfile 构建了一个镜像并将其推送到 Docker 存储库。最后，我们在 Run 阶段运行了一个带有 account-service 应用程序的容器。所有上述阶段都在节点元素的定义之后在 dind-node-1 上执行，节点元素是管道定义中所有其他元素的根。

现在我们可以继续在 Jenkins 的 Web 控制台中定义管道。选择 New Item（新建项目），然后选中 Pipeline 项目类型并输入其名称。确认后应该重定向到管道的配置页面。在该页面中，需要提供 Git 存储库中 Jenkinsfile 的位置，然后设置 SCM 身份验证凭据，如图 14.9 所示。

图 14.9　配置管道

在保存更改之后，管道的配置已准备就绪。要开始构建，可单击 Build Now（立即构建）按钮。在这个阶段有两件事需要澄清。在生产模式中，开发人员可以使用 webhook 机制，该机制由最受欢迎的 Git 主机供应商提供，包括 GitHub、BitBucket 和 GitLab。在将更改推送到存储库后，此机制会自动触发开发人员在 Jenkins 上的构建。为了演示这一点，开发人员必须使用 Docker 在本地运行版本控制系统，如使用 GitLab。还有另一种简

化的测试方法，那就是容器化应用程序直接在 Docker 从属服务器中的 Jenkins 的 Docker 上运行。当然，在正常情况下，开发人员应该在分离的远程机器上启动（该远程机器专门用于部署应用程序）。如图 14.10 所示的就是 Jenkins 的 Web 控制台，它说明了按不同阶段划分的 product-service 服务的构建过程。

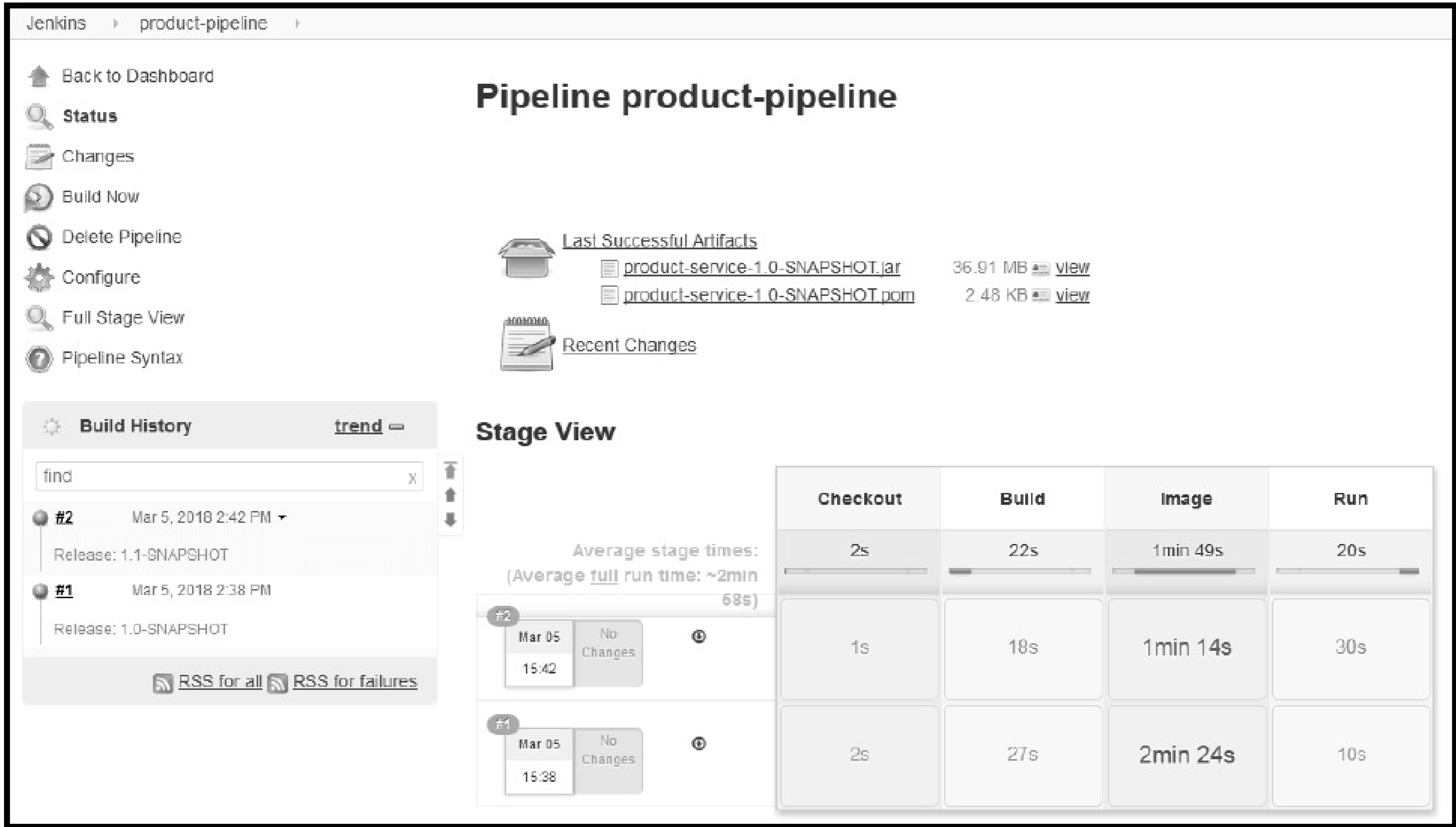



图 14.10　按不同阶段划分的 product-service 服务的构建过程

现在应该为每个微服务创建一个管道，所有创建的管道列表如图 14.11 所示。

| S | W | Name ↓ | Last Success | Last Failure | Last Duration |
|---|---|---|---|---|---|
| | | account-service | 8 min 41 sec - #1 | N/A | 1 min 55 sec |
| | | customer-pipeline | 6 min 20 sec - #1 | N/A | 1 min 56 sec |
| | | discovery-pipeline | 23 min - #1 | N/A | 6 min 6 sec |
| | | order-service | 1 min 58 sec - #1 | N/A | 1 min 54 sec |
| | | product-pipeline | 12 min - #2 | N/A | 2 min 26 sec |

图 14.11　为微服务创建的管道列表

## 14.6　使用 Kubernetes

我们已经在 Docker 容器上启动了示例微服务，甚至使用了 CI 和 CD 自动化管道，以

便在本地计算机上运行它们。但是，开发人员可能会提出一个重要问题：我们如何在更大规模和生产模式下组织我们的环境，使我们能够在多台机器上运行多个容器？实际上，这正是我们根据云原生开发的思想实现微服务时必须要做的。事实证明，在这种情况下仍然存在许多挑战。假设开发人员在多个实例中启动了许多微服务，那么将会有大量容器需要管理。此时的很多操作（例如，在正确的时间启动正确的容器、处理存储的注意事项、向上或向下扩展以及手动处理故障等）对于开发人员来说都将是一场噩梦。幸运的是，有一些平台可以帮助大规模地集群和编排 Docker 容器。目前，走在该领域前列的是 Kubernetes。

Kubernetes 是一个用于管理容器化工作负载和服务的开源平台。它可以充当容器平台、微服务平台、云平台等。它可以自动执行以下操作：跨不同计算机运行容器、向上和向下扩展、在容器之间分配负载，以及在应用程序的多个实例之间保持存储的一致性。它还具有许多其他功能，包括服务发现、负载均衡、配置管理、服务命名和滚动更新等。当然，并非所有这些功能都对开发人员有用，因为 Spring Cloud 也提供了许多类似的功能。

值得一提的是，Kubernetes 不是唯一的容器管理工具。类似的工具还有 Docker Swarm，它是 Docker 中提供的原生工具。但是，由于 Docker 已经宣布了对 Kubernetes 的原生支持，所以它看起来也是一个很自然的选择。在详细讨论任何实际示例之前，开发人员应该知道关于 Kubernetes 的几个重要概念和组件。

## 14.6.1　概念和组件

使用 Kubernetes 时可能需要理解的第一个术语是 pod，它是 Kubernetes 的基本构建块。pod 表示集群中正在运行的进程。它可以由一个或多个容器组成，这些容器保证共存在主机上并将共享相同的资源。每个 pod 一个容器就是最常见的 Kubernetes 用例。每个 pod 在集群中都有唯一的 IP 地址，但部署在同一 pod 中的所有容器都可以通过 localhost 与其他容器进行通信。

另一个常见的组件是服务。服务可以在逻辑上对一组 pod 进行分组，并定义访问它的策略。它有时被称为微服务。默认情况下，服务在集群内部公开，但也可以公开到外部 IP 地址。开发人员可以使用以下 4 种可用行为之一公开服务：ClusterIP、NodePort、LoadBalancer 和 ExternalName。默认选项是 ClusterIP。这会在集群内部 IP 上公开服务，这也使得它只能从集群内部访问。

NodePort 将公开每个节点的 IP 在静态端口上的服务，并自动创建 ClusterIP 以在集群内公开服务。反过来，LoadBalancer 将使用云提供商的负载均衡器在外部公开服务，ExternalName 可以将服务映射到 externalName 字段的内容。这里还应该花点时间来讨论

Kubernetes 的副本控制器（Replication Controller）。它通过在集群中运行指定数量的 pod 副本来处理副本和扩展。如果底层节点出现故障，它还负责替换 pod。Kubernetes 中的每个控制器都是通过 kube-controller-manager 运行的独立进程。开发人员还可以在 Kubernetes 中找到节点控制器（Node Controller）、端点控制器（Endpoint Controller）以及服务账户和令牌控制器。

Kubernetes 使用 etcd 键/值存储作为所有集群数据的后备存储。在集群的每个节点内都有一个名为 kubelet 的代理，它负责确保容器在 pod 中运行。用户发送给 Kubernetes 的每个命令都由 kubeapi-server 公开的 Kubernetes API 处理。

当然，以上只是对 Kubernetes 架构的一个非常简化的解释。有许多组件和工具必须正确配置才能成功运行高可用性 Kubernetes 集群。这不是一项简单的任务，它需要大量有关此平台的知识。幸运的是，有一个工具可以轻松地以本地方式运行 Kubernetes 集群，这个工具就是 Minikube。

## 14.6.2　通过 Minikube 以本地方式运行 Kubernetes

Minikube 是一种工具，可以按本地方式轻松运行 Kubernetes。它在本地计算机上的虚拟机内运行单节点 Kubernetes 集群。它绝对是开发模式中最合适的选择。当然，Minikube 并不支持 Kubernetes 提供的所有功能，只提供了对最重要功能的支持，包括 DNS、NodePorts、Config Map、Dashboard 和 Ingress 等。

要在 Windows 上运行 Minikube，需要安装虚拟化工具。当然，如果开发人员已经运行了 Docker，那么很可能已经安装了 Oracle VM VirtualBox。在这种情况下，除了下载并安装 Minikube 的最新版本之外，不必执行任何其他操作。开发人员可以查看 https://github.com/kubernetes/minikube/releases 和 kubectl.exe，其文本说明的地址为 https://storage.googleapis.com/kubernetes-release/release/stable.txt。文件 minikube.exe 和 kubectl.exe 都应包含在 PATH 环境变量中。此外，Minikube 还提供了自己的安装程序 minikube-installer.exe，它会自动将 minikube.exe 添加到路径中。然后，开发人员可以通过运行以下命令从命令行启动 Minikube。

```
$ minikube start
```

以上命令将初始化一个名为 minikube 的 kubectl 上下文。它包含允许开发人员与 Minikube 集群通信的配置。现在可以使用 kubectl 命令来维护由 Minikube 创建的本地集群，并在那里部署容器。命令行界面的替代解决方案是 Kubernetes 仪表板。可以通过调用 minikube 仪表板为节点启用 Kubernetes 仪表板。开发人员可以使用此仪表板创建、更

新或删除部署，以及列出和查看所有 pod、服务、入口和副本控制器的配置。通过调用以下命令可以轻松地停止和删除本地集群。

```
$ minikube stop
$ minikube delete
```

## 14.6.3　部署应用程序

Kubernetes 集群上现有的每个配置都由 Kubernetes 对象表示。这些对象可以通过 Kubernetes API 进行管理，并且应该以 YAML 格式表示。开发人员可以直接使用该 API，但也可以利用 kubectl 命令行界面进行所有必要的调用。Kubernetes 中新创建的对象的描述必须提供描述其所需状态的规范，以及有关该对象的一些基本信息。以下是应始终设置的 YAML 配置文件中的一些必填字段。

- apiVersion：这表示用于创建对象的 Kubernetes API 的版本。API 总是在请求中需要 JSON 格式，但 kubectl 会自动将 YAML 输入转换为 JSON。
- kind：设置要创建的对象的类型。有一些预定义类型可用，如 Deployment、Service、Ingress 或 ConfigMap。
- metadata：这允许开发人员通过名称、UID 或可选命名空间来标识对象。
- spec：这是对象的正确定义。规范的精确格式取决于对象的类型，并包含特定于该对象的嵌套字段。

一般来说，在 Kubernetes 上创建新对象时，其 kind 是 Deployment。在如下所示的 Deployment YAML 文件中，有两个重要的字段设置。第一个字段是 replicas，它指定了所需 pod 的数量。实际上，这意味着开发人员运行容器化应用程序的两个实例。第二个字段是 spec.template.spec.containers.image，它设置将在 pod 中启动的 Docker 镜像的名称和版本。该容器将在端口 8090 上公开，而 order-service 服务也将在端口 8090 上侦听 HTTP 连接。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: order-service
spec:
  replicas: 2
  selector:
    matchLabels:
      app: order-service
```

```
  template:
    metadata:
      labels:
        app: order-service
    spec:
      containers:
      - name: order-service
        image: piomin/order-service:1.0
        env:
        - name: EUREKA DEFAULT ZONE
          value: http://discovery-service:8761/eureka
        ports:
        - containerPort: 8090
          protocol: TCP
```

假设上述代码存储在文件 order-deployment.yaml 中，那么开发人员就可以使用命令管理方式在 Kubernetes 上部署自己的容器化应用程序，如下所示。

```
$ kubectl create -f order-deployment.yaml
```

或者，也可以基于声明式管理方式执行相同的操作，如下所示。

```
$ kubectl apply -f order-deployment.yaml
```

现在开发人员必须为所有微服务和 discovery-service 服务创建相同的部署文件。discovery-service 服务的客体是一个非常希望了解的事项。开发人员可以选择使用基于 pod 和服务的内置 Kubernetes 发现，但这里的主要目标是在该平台上部署和运行 Spring Cloud 组件，因此，在部署任何微服务之前，首先应该在 Kubernetes 上部署、运行和公开 Eureka。以下是 discovery-service 的部署文件，它也可以通过调用 kubectl apply 命令应用于 Kubernetes。

```
apiVersion: apps/v1
kind: Deployment
metadata:
 name: discovery-service
 labels:
  run: discovery-service
spec:
 replicas: 1
 selector:
  matchLabels:
   app: discovery-service
 template:
```

```
  metadata:
   labels:
    app: discovery-service
  spec:
   containers:
   - name: discovery-service
     image: piomin/discovery-service:1.0
   ports:
   - containerPort: 8761
     protocol: TCP
```

如果创建的是 Deployment 类型的对象，则 Kubernetes 会自动创建 pod。它们的数量等于 replicas 字段中设置的值。pod 无法公开由部署在容器上的应用程序提供的 API，它只表示集群上正在运行的进程。要访问在 pod 中运行的微服务提供的 API，必须定义一个服务。那么这里的服务是什么？服务是一种抽象，它定义了一组逻辑 pod 和一个访问它们的策略。服务所针对的一组 pod 通常由标签选择器确定。Kubernetes 提供了 4 种服务类型，最简单和默认的是 ClusterIP，它在内部公开服务。如果要从集群外部访问服务，则应定义 NodePort 类型。此选项已在以下示例 YAML 文件中列出。现在，所有微服务都可以使用其 Kubernetes 服务名称与 Eureka 进行通信。

```
apiVersion: v1
kind: Service
metadata:
 name: discovery-service
  labels:
   app: discovery-service
spec:
 type: NodePort
 ports:
   - protocol: TCP
     port: 8761
     targetPort: 8761
 selector:
   app: discovery-service
```

事实上，部署在 Minikube 上的所有微服务都应该在集群外部使用，因为开发人员希望访问它们公开的 API。为此，需要提供与前面示例中类似的 YAML 配置，当然也别忘记更改服务的名称、标签和端口。

在我们的架构中，现在应该只剩下最后一个组件还未介绍：API 网关。开发人员当然可以使用 Zuul 代理部署容器，但是这里我们想要介绍另一个流行的 Kubernetes 对象：

Ingress。该组件负责管理通过 HTTP 公开的服务的外部访问。Ingress 可以提供负载均衡、SSL 终端和基于名称的虚拟主机。Ingress 配置 YAML 文件如下所示。请注意，可以在不同 URL 路径上的同一端口 80 上访问所有服务。

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
 name: gateway-ingress
spec:
 backend:
  serviceName: default-http-backend
  servicePort: 80
 rules:
 - host: microservices.example.pl
   http:
   paths:
   - path: /account
     backend:
       serviceName: account-service
       servicePort: 8091
   - path: /customer
     backend:
       serviceName: customer-service
       servicePort: 8092
   - path: /order
     backend:
       serviceName: order-service
       servicePort: 8090
   - path: /product
     backend:
       serviceName: product-service
       servicePort: 8093
```

## 14.6.4　维护集群

维护 Kubernetes 集群相当复杂。本节将演示如何使用一些基本命令和用户界面仪表板来查看集群上当前存在的对象。这里不妨列出为运行基于微服务的示例系统而创建的元素。要完成此操作，首先可以通过运行 kubectl get deployments 命令显示部署列表，这将导致如图 14.12 所示的结果。

一个部署可以创建许多 pod。开发人员可以通过调用 kubectl get pods 命令来检查 pod

列表，如图 14.13 所示。

Deployments

| Name | Labels | Pods | Age | Images |
|---|---|---|---|---|
| product-service | app: product-service | 1 / 1 | 12 minutes | piomin/product-service:1.0 |
| order-service | app: order-service | 2 / 2 | 16 minutes | piomin/order-service:1.0 |
| account-service | app: account-service | 1 / 1 | an hour | piomin/account-service:1.0 |
| discovery-service | run: discovery-service | 1 / 1 | an hour | piomin/discovery-service:1.0 |

图 14.12　显示部署列表

```
C:\Users\minkowp>kubectl get pods
NAME                                READY     STATUS    RESTARTS   AGE
account-service-56fdc7cf79-nkrfn    1/1       Running   0          3h
discovery-service-dcdb86b7b-sz15b   1/1       Running   1          4h
order-service-796bbf6975-fbd9z      1/1       Running   0          3h
order-service-796bbf6975-z45f6      1/1       Running   0          3h
product-service-78dd7b555c-rnmkp    1/1       Running   0          2h
```

图 14.13　通过调用 kubectl get pods 命令来检查 pod 列表

可以通过用户界面仪表板来查看相同的列表。开发人员可以通过单击所选行来查看这些详细信息，或者通过单击每行右侧的可用图标来查看容器日志，如图 14.14 所示。

Pods

| Name | Node | Status | Restarts | Age |
|---|---|---|---|---|
| product-service-78dd7b555c-rnmkp | minikube | Running | 0 | 12 minutes |
| order-service-796bbf6975-fbd9z | minikube | Running | 0 | 16 minutes |
| order-service-796bbf6975-z45f6 | minikube | Running | 0 | 16 minutes |
| account-service-56fdc7cf79-nkrfn | minikube | Running | 0 | an hour |
| discovery-service-dcdb86b7b-szl5b | minikube | Running | 1 | an hour |

图 14.14　通过用户界面仪表板来查看 pod 列表

可以使用命令 kubectl get services 显示可用服务的完整列表，如图 14.15 所示。这里有一些有趣的字段，包括一个表示集群内可用服务的 IP 地址（CLUSTER-IP）和一对端口（PORT(S)），它们的服务将在内部和外部公开。还可以在地址 http://192.168.99.100:31099 上调用由 account-service 服务公开的 HTTP API，或者在地址 http://192.168.99.100:31931 上访问 Eureka 用户界面仪表板。

```
C:\Users\minkowp>kubectl get services
NAME                TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)          AGE
account-service     NodePort    10.111.189.70   <none>        8091:31099/TCP   3h
discovery-service   NodePort    10.98.219.32    <none>        8761:31931/TCP   3h
kubernetes          ClusterIP   10.96.0.1       <none>        443/TCP          4h
order-service       NodePort    10.98.218.69    <none>        8090:30837/TCP   3h
product-service     NodePort    10.98.83.89     <none>        8093:30565/TCP   2h
```

图 14.15　使用命令 kubectl get services 显示可用服务的完整列表

与以前的对象类似，也可以使用 Kubernetes 仪表板来显示服务，如图 14.16 所示。

Services

| Name | Labels | Cluster IP | Internal endpoints | External endpoints | Age |
|---|---|---|---|---|---|
| product-service | app: product-service | 10.98.83.89 | product-service:8093 TCP<br>product-service:30565 TCP | - | 0 seconds |
| order-service | app: order-service | 10.98.218.69 | order-service:8090 TCP<br>order-service:30837 TCP | - | 17 minutes |
| account-service | app: account-service | 10.111.189.70 | account-service:8091 TCP<br>account-service:31099 TCP | - | 28 minutes |
| discovery-service | app: discovery-service | 10.98.219.32 | discovery-service:8761 TCP<br>discovery-service:31931 TCP | - | 47 minutes |
| kubernetes | component: apiserver<br>provider: kubernetes | 10.96.0.1 | kubernetes:443 TCP<br>kubernetes:0 TCP | - | an hour |

图 14.16　使用 Kubernetes 仪表板显示的服务列表

## 14.7　小　　结

本章讨论了许多与 Spring Cloud 无明显关联的主题，但本章中介绍的工具将允许开发人员充分利用迁移到基于微服务的架构的优势。当使用 Docker、Kubernetes 或持续集成/持续交付（CI/CD）工具时，使用 Spring Cloud 进行云原生开发具有明显的优势。当然，所有提供的示例都已在本地计算机上启动，但开发人员可以参考这些示例来设想如何在跨远程计算机集群的生产环境中设计该过程。

本章向开发人员演示了如何简单快捷地从在本地计算机上手动运行 Spring 微服务转移到从源代码构建应用程序的全自动化流程，使用应用程序创建和运行 Docker 镜像，并将其部署在由多台计算机组成的集群上。想要在同一章中描述 Docker、Kubernetes 和 Jenkins 等复杂工具提供的所有功能并不容易，所以，本章的主要目的是让开发人员了解如何基于容器化、自动部署、扩展和私有云等概念设计和维护现代架构的宏观知识。

我们现在已经接近本书的结尾了。我们已经讨论了与 Spring Cloud 框架相关的大多数计划主题。第 15 章将展示如何使用 Web 上可用的两个最流行的云平台，从而允许开发人员持续交付 Spring Cloud 应用程序。

# 第 15 章　云平台上的 Spring 微服务

Pivotal 公司将 Spring Cloud 定义为加速云原生应用程序开发的框架。今天，当我们谈论云原生应用程序时，首先想到的是快速交付软件的能力。为了满足这些需求，开发人员应该能够快速构建可扩展、可移植且准备频繁更新的新应用程序和设计架构。提供容器化和编排机制的工具有助于开发人员建立和维护这样的架构。实际上，本书前面章节中已经讨论过 Docker 或 Kubernetes 等工具，它们都允许开发人员创建自己的私有云并在其上运行 Spring Cloud 微服务。虽然应用程序不必部署在公共云上，但它包含云软件的所有最重要的特征。

在公共云上部署 Spring 应用程序只是一种可能性，而不是必需的。但是，有一些非常有趣的云平台允许开发人员在几分钟内轻松运行微服务并在网络上公开它们。其中一个平台就是 Pivotal Cloud Foundry（PCF），它优于其他平台的优势在于它对 Spring Cloud 服务的原生支持，包括使用 Eureka、Config Server 和 Hystrix 断路器进行发现。开发人员还可以通过启用 Pivotal 提供的代理服务轻松设置完整的微服务环境。

本章将要介绍的另一个云平台是 Heroku。与 PCF 相比，它不支持任何编程框架。Heroku 是一个完全托管的多语言平台，可以让开发人员快速交付软件。一旦推送了对存储在 GitHub 存储库中的源代码的更改，它就可以自动构建和运行应用程序。它还提供了许多附加服务，可以使用单个命令进行配置和扩展。

本章将要讨论的主题包括：

- Pivotal Web Services 平台简介。
- 使用 CLI、Maven 插件和用户界面仪表板在 Pivotal Cloud Foundry 上部署和管理应用程序。
- 使用 Spring Cloud Foundry 库准备应用程序以使其在平台上正常工作。
- 在 Heroku 平台上部署 Spring Cloud 微服务。
- 管理代理服务。

## 15.1　Pivotal Cloud Foundry

虽然 Pivotal 平台可以运行用多种语言编写的应用程序，如 Java、.NET、Ruby、

JavaScript、Python、PHP 和 Go 等，但它对 Spring Cloud Services 和 Netflix OSS 工具提供了最好的支持。这非常有意义，因为他们是开发 Spring Cloud 的人。图 15.1 说明了 Pivotal Cloud 平台提供的基于微服务的架构(该图的英文版也可以在 Pivotal 的官方网站上找到)。开发人员可以在 Cloud Foundry 上使用 Spring Cloud 快速利用常见的微服务模式，包括分布式配置管理、服务发现、动态路由、负载均衡和容错等。

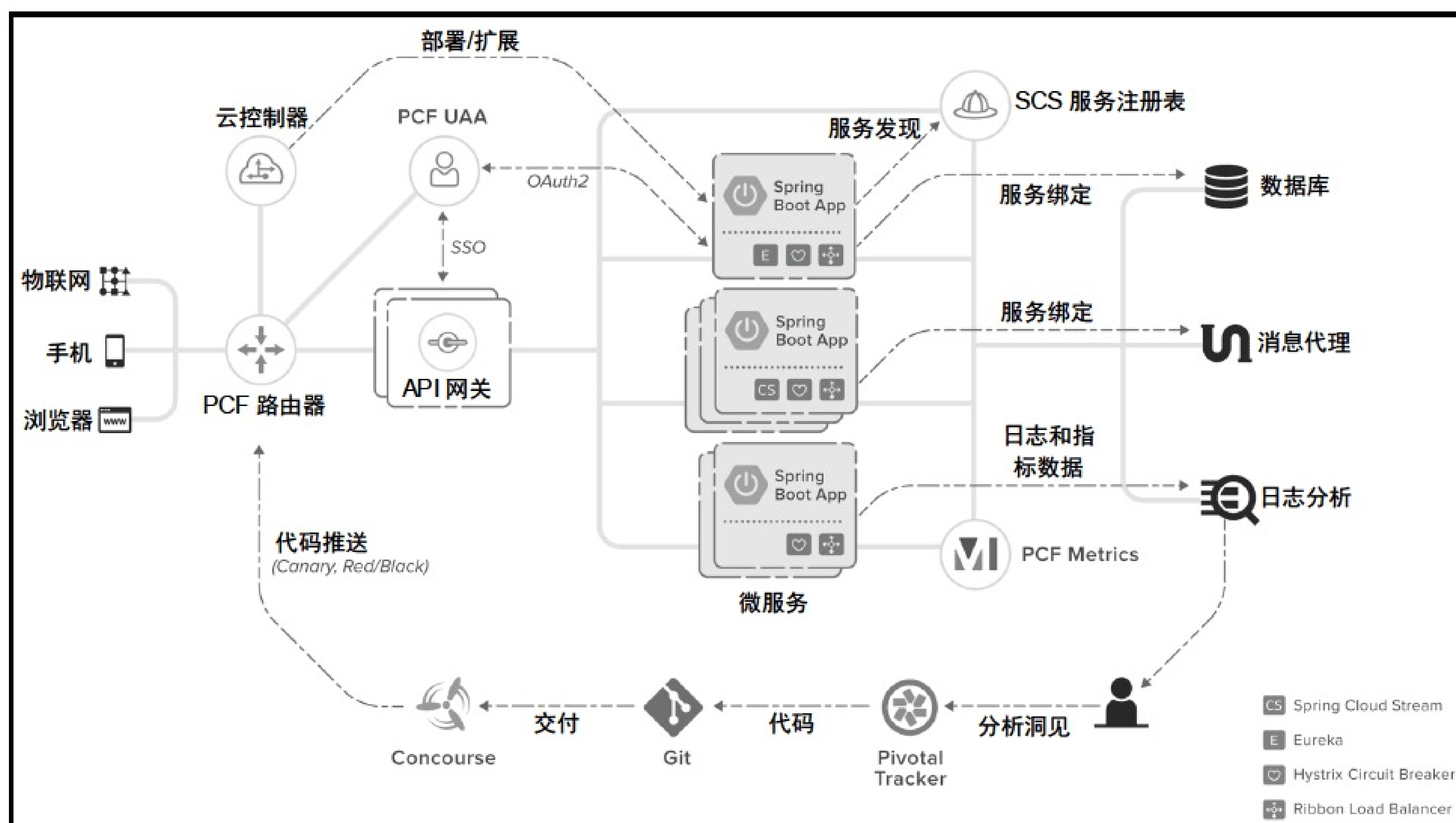



图 15.1　Pivotal Cloud 平台提供的基于微服务的架构

## 15.1.1　使用模式

开发人员可以按 3 种不同的模式使用 Pivotal 平台。这些模式是根据主机进行区分的，而主机就是部署应用程序的位置。以下是可用解决方案的列表。

- PCF Dev：Pivotal 平台的这个实例可以在一台虚拟机上以本地方式运行。它专为实验和开发需求而设计。它不提供所有可能的功能和服务。例如，只有一些诸如 Redis、MySQL 和 RabbitMQ 之类的内置服务。但是，PCF Dev 还支持 Spring Cloud Services（SCS）以及完整版 PCF 中支持的所有语言。值得注意的是，如果开发人员想要以本地方式运行包含 SCS 的 PCF Dev，则需要超过 6 GB 的内存。
- Pivotal Web Services：这是一个可在线访问的云原生平台，网址为 https://run.pivotal.io/。它就像 Pivotal Cloud Foundry 一样，提高托管功能，并且按小时付费。

它不提供 Pivotal Cloud Foundry 中提供的所有功能和服务。例如，开发人员可以只启用由 Pivotal 的 SaaS 合作伙伴提供的服务。Pivotal Web Services 最适合初创公司或个人团队。我们将在本章后面的小节中使用此 Pivotal 平台托管模型进行演示。

- Pivotal Cloud Foundry：这是一个功能齐全的云原生平台，可在任何主要的公共 IaaS 上运行，包括 AWS、Azure 和 Google Cloud Platform，或者基于 OpenStack 或 VMware vSphere 的私有云。它是适用于大型企业环境的商业解决方案。

## 15.1.2　准备应用程序

由于 Pivotal Web Services 对 Spring Cloud 应用程序具有原生支持，因而部署过程非常简单。但是，它确实需要在应用程序端具有特定的依赖项和配置——特别是如果开发人员的微服务必须与 Pivotal 平台（如 Service Registry、Config Server 或 Circuit Breaker）提供的内置服务集成的话，更是如此。除了 Spring Cloud 的标准依赖项管理之外，还应该在 pom.xml 中包含 spring-cloud-services-dependencies，最新版本与 Edgware.SR2 版本列车一起使用，如下所示。

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>Edgware.SR2</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
        <dependency>
            <groupId>io.pivotal.spring.cloud</groupId>
            <artifactId>spring-cloud-services-dependencies</artifactId>
            <version>1.6.1.RELEASE</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

根据所选的集成服务，开发人员可能希望在项目中包含以下工件。我们决定使用 Pivotal 平台提供的所有 Spring Cloud 功能，因此，我们的微服务将获取配置服务器的属性，在 Eureka 中注册它们，并使用 Hystrix 命令包装服务间通信。

以下是为在 Pivotal 平台上部署的应用程序启用发现客户端、配置客户端和断路器所需要的依赖项。

```
<dependency>
    <groupId>io.pivotal.spring.cloud</groupId>
    <artifactId>spring-cloud-services-starter-circuit-breaker
    </artifactId>
</dependency>
<dependency>
    <groupId>io.pivotal.spring.cloud</groupId>
    <artifactId>spring-cloud-services-starter-config-client</artifactId>
</dependency>
<dependency>
    <groupId>io.pivotal.spring.cloud</groupId>
    <artifactId>spring-cloud-services-starter-service-registry</artifactId>
</dependency>
```

我们将为示例微服务提供更多集成。所有这些都将在 MongoDB 中存储数据，而 MongoDB 也可以作为 Pivotal 平台上的服务提供。要完成此目标，首先应该在项目依赖项中包含 starter spring-boot-starter-data-mongodb。

```
<dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-data-mongodb</artifactId>
</dependency>
```

应该使用 spring.data.mongodb.uri 属性在配置设置中提供 MongoDB 数据库的地址。为了允许应用程序与 MongoDB 数据库连接，我们必须创建一个 Pivotal 的服务 mLab，然后将其绑定到应用程序。默认情况下，与绑定服务相关的元数据将作为环境变量 $VCAP_SERVICES 公开给应用程序。这种方法的主要动机是 Cloud Foundry 被设计为多语言，这意味着任何语言和平台都可以作为构建包（Buildpack）获得支持。可以使用 vcap 前缀注入所有 Cloud Foundry 属性。如果想要访问 Pivotal 的服务，则应该使用 vcap.services 前缀，然后传递如下所示的服务名称。

```
spring:
 data:
  mongodb:
   uri: ${vcap.services.mlab.credentials.uri}
```

实际上，这就是需要在应用程序端完成的所有工作，这样就可以使它们与 Pivotal 平台上创建的组件一起正常工作。现在我们必须与在 Spring 中编写的标准微服务相同的方

式启用 Spring Cloud 功能，如下所示。

```
@SpringBootApplication
@EnableDiscoveryClient
@EnableFeignClients
@EnableCircuitBreaker
public class OrderApplication {

    public static void main(String[] args) {
        SpringApplication.run(OrderApplication.class, args);
    }

}
```

### 15.1.3　部署应用程序

可以通过 3 种不同的方式在 Pivotal Web Service（PWS）平台上管理应用程序。第一种方式是通过 https://console.run.pivotal.io 上提供的 Web 控制台。开发人员可以通过这种方式监控、扩展、重新启动已部署的应用程序，启用和禁用服务，定义新指标以及更改账户设置。但是，使用 Web 控制台（换句话说，也就是初始应用程序部署）却无法执行此操作，它需要使用命令行界面（Command-Line Interface，CLI）执行。开发人员可以从 pivotal.io 网站下载所需的安装程序。安装完成之后，即可通过输入 cf 来调用计算机上的 Cloud Foundry CLI，如 cf help。

#### 1．使用命令行界面

命令行界面提供了一组命令，允许开发人员在 Cloud Foundry 上管理应用程序、代理的服务、空间、域和其他组件等。接下来将介绍在 PWS 上运行应用程序时应该知道的最重要的命令。

（1）要部署应用程序，必须先导航到其目录。然后，应该使用 cf login 命令登录 PWS，如下所示。

```
$ cf login -a https://api.run.pivotal.io
```

（2）使用 cf push 命令将应用程序推送到 Pivotal Web Service，并传递服务的名称。

```
$ cf push account-service -p target / account-service-1.0.0-
SNAPSHOT.jar
```

（3）或者，开发人员也可以在应用程序的根目录中提供 manifest.yml 文件以及所有

必需的部署设置。在这种情况下，开发人员只需运行 cf push 命令而无须任何其他参数，如下所示。

```
---
applications:
- name: account-service
  memory: 300M
  random-route: true
  path: target/account-service-1.0-SNAPSHOT.jar
```

（4）使用如上例所示的 manifest.yml 文件中提供的配置设置进行部署将失败。要查看原因，可以运行命令 cf logs。原因是堆的内存限制不足。

```
$ cf logs account-service --recent
```

默认情况下，平台将为代码缓存分配 240MB，为元空间（Metaspace）分配 140MB，为每个线程分配 1MB，并假设 Tomcat 连接器最多有 200 个线程。这样就很容易计算出，使用这些设置，每个应用程序需要大约 650MB 的分配内存（而上面的示例仅分配了 300MB）。我们可以通过调用 cf set-env 命令并传递 JAVA_OPTS 参数来更改这些设置，如以下代码所示。像这样的内存限制在生产模式下是不够的，但可以用于测试目的。要确保这些更改生效，可以使用 cf restage 命令，如下所示。

```
$ cf set-env account-service JAVA_OPTS "-Xmx150M -Xss250K -
XX:ReservedCodeCacheSize=70M -XX:MaxMetaspaceSize=90M"
$ cf restage account-service
```

分配的内存很重要，特别是如果只有 2GB 内存可用于免费账户。应用默认内存设置后，我们就只能在 Pivotal 平台上部署两个应用程序，因为每个应用程序占用 1GB 的内存。虽然已经解决了前面描述的问题，但我们的应用仍然无法正常工作。

### 2. 绑定到服务

在引导期间，应用程序无法连接所需的服务。出现此问题的原因是服务未默认绑定到应用程序。可以通过运行命令 cf services 来显示在空间中创建的所有服务，并通过调用命令 cf bind- service 将它们中的每一个绑定到给定的微服务。在以下示例命令的执行中，我们将 Eureka、配置服务器和 MongoDB 都绑定到 account-service 服务。最后，可以再次运行 cf restage，此时一切都应该正常工作，如下所示。

```
$ cf bind-service account-service discovery-service
$ cf bind-service account-service config-service
$ cf bind-service account-service sample-db
```

### 3．使用 Maven 插件

如前文所述，命令行界面和 Web 控制台并不是在 Pivotal 平台上管理应用程序的唯一方法。Cloud Foundry 团队已经实现了 Maven 插件，以促进和加快应用程序部署。有趣的是，同一个插件可用于管理任何 Cloud Foundry 实例的推送和更新，而不仅限于由 Pivotal 提供的实例。

使用 Cloud Foundry 的 Maven 插件时，开发人员可以轻松地将云部署集成到 Maven 项目的生命周期中。这允许开发人员在 Cloud Foundry 中推送、删除和更新项目。如果想要将项目与 Maven 一起推送，只需运行以下命令。

```
$ mvn clean install cf:push
```

一般来说，Maven 插件提供的命令与命令行界面提供的命令非常相似。例如，开发人员可以通过执行命令 mvn cf:apps 来显示应用程序的列表。如果要删除某个应用程序，则可以运行以下命令。

```
$ mvn cf:delete -Dcf.appname = product-service
```

如果要将某些更改上传到现有应用程序，则可以使用 cf:update，其命令如下。

```
$ mvn clean install cf:update
```

在运行任何命令之前，都必须正确配置插件。首先，需要传递 Cloud Foundry 登录凭据。建议将它们分别存储在 Maven 的 settings.xml 文件中。服务器标记内的典型条目可能如下所示。

```
<settings>
    ...
    <servers>
        <server>
            <id>cloud-foundry-credentials</id>
            <username>piotr.minkowski@play.pl</username>
            <password>123456</password>
        </server>
    </servers>
    ...
</settings>
```

使用 Maven 插件而不是命令行界面的命令有一个重要的优点：开发人员可以在一个位置配置所有必要的配置设置，并可以在应用程序构建期间使用单个命令应用它们。插件的完整配置显示在以下代码段中。除了一些基本设置（包括空间、内存和大量实例）

之外，还可以使用 JAVA_OPTS 环境变量更改内存限制，并将所需服务绑定到应用程序。运行 cf:push 命令之后，就可以在 https://product-service-piomin.cfapps.io/地址使用 product-service 服务。

```
<plugin>
    <groupId>org.cloudfoundry</groupId>
    <artifactId>cf-maven-plugin</artifactId>
    <version>1.1.3</version>
    <configuration>
        <target>http://api.run.pivotal.io</target>
        <org>piotr.minkowski</org>
        <space>development</space>
        <appname>${project.artifactId}</appname>
        <memory>300</memory>
        <instances>1</instances>
        <server>cloud-foundry-credentials</server>
        <url>https://product-service-piomin.cfapps.io/</url>
        <env>
            <JAVA_OPTS>-Xmx150M -Xss250K -XX:ReservedCodeCacheSize=70M -
XX:MaxMetaspaceSize=90M</JAVA_OPTS>
        </env>
        <services>
            <service>
                <name>sample-db</name>
                <label>mlab</label>
                <plan>sandbox</plan>
            </service>
            <service>
                <name>discovery-service</name>
                <label>p-service-registry</label>
                <plan>standard</plan>
            </service>
            <service>
                <name>config-service</name>
                <label>p-config-server</label>
                <plan>standard</plan>
            </service>
        </services>
    </configuration>
</plugin>
```

### 15.1.4　维护

假设开发人员已经成功部署了构建基于微服务的示例系统的所有应用程序，那么现在就可以使用 Pivotal Web Services 仪表板甚至命令行界面的命令轻松管理和监视它们。Pivotal 平台提供的免费试用版为开发人员提供了许多维护应用程序的可能性和工具，接下来我们将介绍一些有趣的功能。

#### 1．访问部署的详细信息

开发人员可以通过运行命令 cf apps 或导航到 Web 控制台中我们空间的主站点来列出所有已部署的应用程序。在如图 15.2 所示的屏幕截图中即可看到该列表。表的每一行代表一个应用程序。除了名称之外，还有关于其状态、实例数、分配的内存、部署时间以及平台外可用服务的 URL 信息等。如果在应用程序部署期间未指定 URL 地址，则这些 URL 会自动生成。

图 15.2　访问部署的详细信息

可以通过单击每一行来了解有关该应用程序的详细信息，也可以在命令行界面中使用命令 cf app <app-name> 或 cf app order-service 访问类似信息。图 15.3 显示了应用程序详细信息视图的主面板，其中包含每个实例的事件历史记录、摘要，以及内存、磁盘和 CPU 使用情况。在此面板中，可以通过单击 Scale（扩展）按钮来扩展应用程序。还有其他几个选项卡可用，通过切换到其中一个即可执行相应操作。例如，单击 Services（服务）可以检查所有绑定的服务，单击 Route（路由）可以分配外部 URL，单击 Logs（日志）可以显示日志，单击 Trace（跟踪）可以查看传入请求的历史记录。

当然，开发人员始终可以使用命令行界面收集与上一示例中所示相同的详细信息。如果执行 cf logs <app-name>命令，则会附加到由应用程序生成的 stdout。开发人员还可以使用绑定应用程序列表显示已激活的 Pivotal 托管服务列表，如图 15.4 所示。

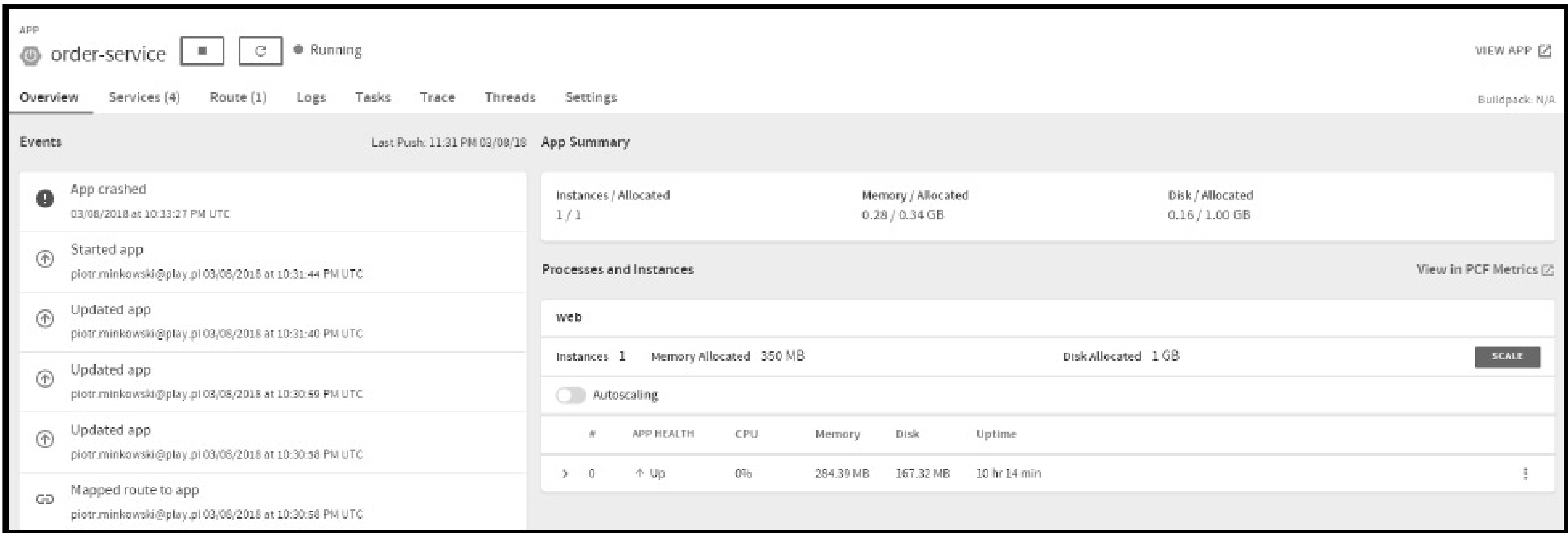



图 15.3　查看有关应用程序的详细信息

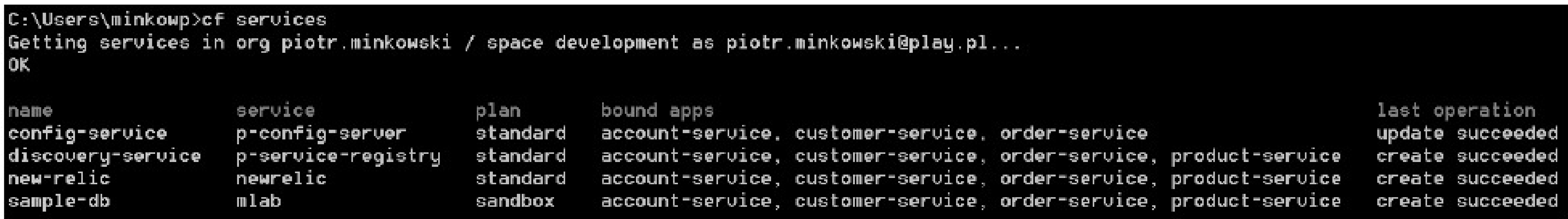



图 15.4　显示已激活的 Pivotal 托管服务列表

## 2．管理应用程序生命周期

Pivotal Web Services 提供的另一个非常有用的功能是管理应用程序生命周期的能力。换句话说，只需单击一下，我们就可以轻松地停止、启动和重新启动应用程序。在执行请求命令之前，系统将出现提示要求确认，如图 15.5 所示。

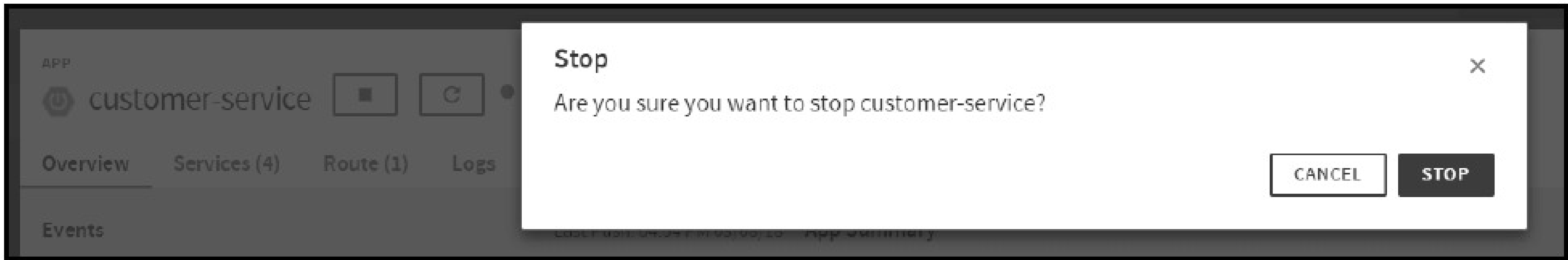



图 15.5　系统提示要求确认操作

运行以下命令行界面的命令之一可以实现相同的结果。

```
$ cf stop <app-name>
$ cf restart <app-name>
$ cf start <app-name>
```

## 3．扩展

使用云解决方案的最重要的原因之一是能够轻松扩展应用程序。Pivotal 平台以非常

直观的方式处理这些问题。首先，开发人员可以决定在每个部署阶段启动应用程序的实例数。例如，如果决定使用 manifest.yml 并通过 cf push 命令部署它，则创建的实例数将由字段实例确定，如以下代码段所示。

```
---
applications:
- name: account-service
  memory: 300M
  instances: 2
  host: account-service-piomin
  domain: cfapps.io
  path: target/account-service-1.0-SNAPSHOT.jar
```

可以在启动的应用程序上修改正在运行的实例数以及内存和 CPU 限制。实际上，有两种可用的扩展方法。开发人员既可以手动设置应启动的实例数，也可以启用自动扩展，在自动扩展中，只需根据选定指标的阈值定义一个条件即可。Pivotal 平台上的自动扩展功能将通过一个名为 PCF App Autoscaler 的工具实现。开发人员可以从以下 5 个可用规则中进行选择，具体如下。

- CPU 利用率
- 内存利用率
- HTTP 延迟
- HTTP 吞吐量
- RabbitMQ 深度

开发人员可以定义多个活动规则。对于向下扩展（Scale Down）来说，这些规则中的每一个都具有单个指标的最小值；而对于向上扩展（Scale Up）来说，自然就是最大值了。在图 15.6 中，显示了 customer-service 服务的自动扩展设置。在这里，我们决定应用的是 HTTP Throughput（HTTP 吞吐量）和 HTTP Latency（HTTP 延迟）规则。如果 99%流量的延迟低于 20ms（毫秒），则应该禁用应用程序的一个实例，因为连接速度足够，所以不必出现多个实例；反过来，如果延迟大于 200ms（毫秒），则平台应该再附加一个实例，以提高处理能力。

开发人员还可以手动控制运行实例的数量。自动扩展具有许多优点，但手动方法可以让开发人员更好地控制该过程。由于每个应用程序的内存有限，因而仍有其他实例的空间。在我们的示例系统中，重载（Overload）最多的应用程序是 account-service 服务，因为它在订单创建期间需要调用，在订单确认时又需要调用。所以，我们可以给它再添加一个实例。要执行此操作，可以转到 account-service 服务详细信息面板，然后单击

Processes and Instances（进程和实例）下的 Scale（扩展）。在出现 Scale app（扩展程序）设置界面时，在 Instances（实例）框中输入目标实例数，然后单击 APPLY CHANGES（应用更改）按钮，如图 15.7 所示。经过这样的修改之后，开发人员应该看到两个可用的 account-service 服务实例。



图 15.6　自动扩展规则设置



图 15.7　手动控制运行实例的数量

### 4．提供代理服务

我们已经了解了如何使用 cf bind-service 命令和 Maven 插件将应用程序绑定到服务。但是，我们现在应该看看如何启用和配置服务。开发人员可以轻松显示所有可用服务的列表，然后使用 Pivotal 的仪表板启用它们，这可以在 Marketplace（市场）下找到。

使用 Pivotal Web Services 提供代理服务非常容易。在安装之后，某些服务已可供使用，而无须任何其他配置。我们所要做的就是将它们绑定到选定的应用程序，并在应用程序的设置中正确传递其网络地址。通过用户界面仪表板可以轻松地将每个应用程序绑

定到服务。首先，导航到服务的主页面。在那里，开发人员将看到当前绑定的应用程序列表。可以先单击 BIND APP（绑定应用程序）按钮，然后从显示的列表中选择一个程序来将新应用程序绑定到服务，如图 15.8 所示。

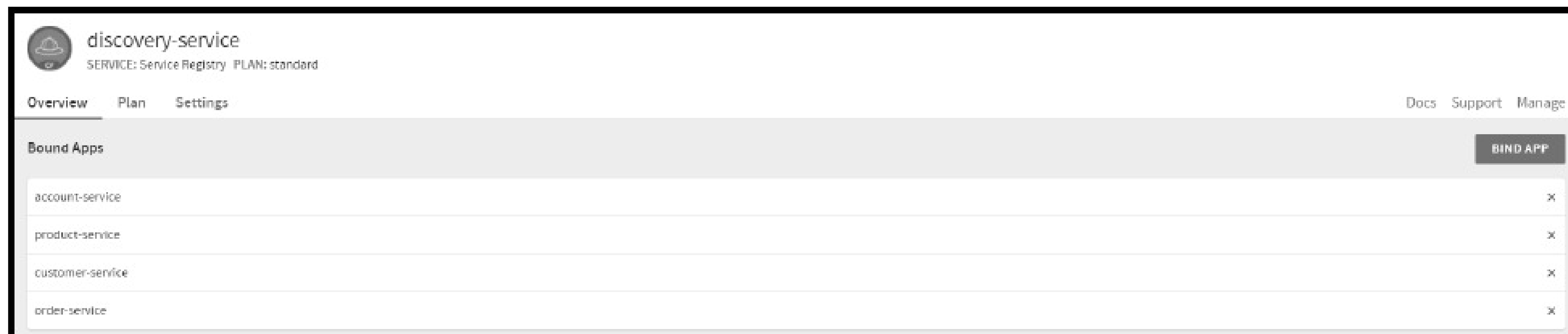



图 15.8　绑定应用程序

除了在 Marketplace（市场）中启用注册表服务并将其绑定到应用程序以便在 Pivotal Web 服务上启用发现功能之外，开发人员不必执行任何其他操作。当然，如果需要，也可以在客户端覆盖某些配置设置。已注册的应用程序的完整列表可以显示在服务主配置面板中 Manage（管理）下的 Eureka 仪表板中。在图 15.9 中可以看到，有两个正在运行的 account-service 服务实例，这是因为之前已对其进行了扩展，但是其他微服务仍然只有一个运行实例。

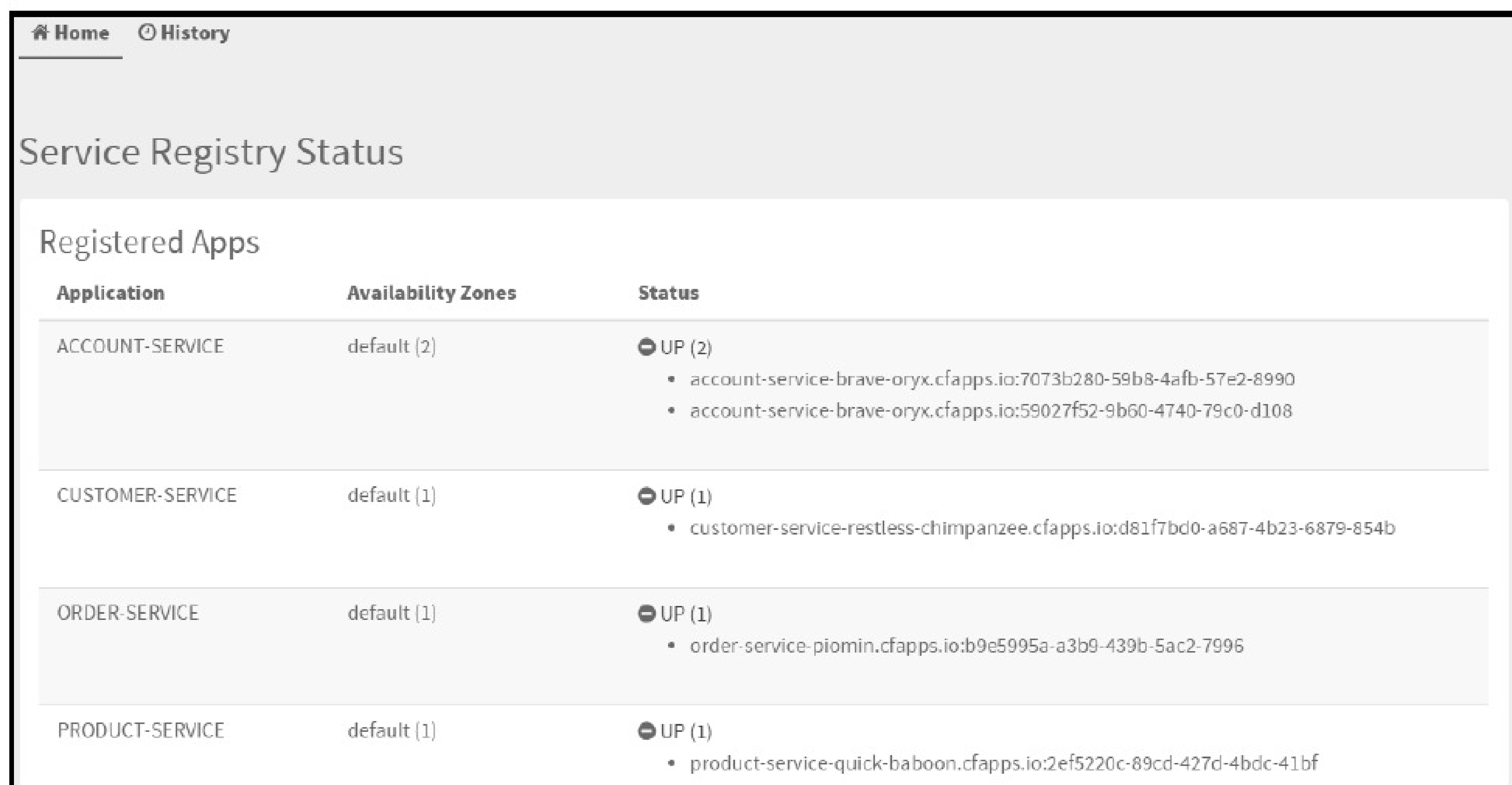



图 15.9　查看已注册的应用程序的完整列表

与发现服务相比，配置服务器需要包含其他设置。和以前一样，开发人员应该导航

到其主面板，然后选择 Manage（管理）。在这里，开发人员将被重定向到配置表单，并且在那里必须提供配置参数作为 JSON 对象。count 参数指定供应所需的节点数，如果实例可以升级则选中 Upgrade（升级）选项，而 force 参数则会强制升级，即使当前实例已经是最新可用的版本。其他配置参数取决于用于存储属性源的后端类型。关于存储属性源的后端类型，在本书第 5 章“使用 Spring Cloud Config 进行分布式配置”中已经详细介绍过，最流行的 Spring Cloud Config Server 解决方案是基于 Git 存储库的。我们在 GitHub 上创建了一个示例存储库，其中提交了所有必需的源。以下是应在 Pivotal Web Services 上为 Config Server 提供的 JSON 格式的参数。

```
{
    "count": 1,
    "git": {
        "password": "****",
        "uri":
        "https://github.com/piomin/sample-spring-cloud-pcf-config.git",
        "username": "piomin"
    }
}
```

我们提供的示例应用程序使用的最后一个代理服务托管了一个 MongoDB 实例。在该服务的主面板上导航到 Manage（管理）之后，开发人员应该被重定向到 https://mlab.com/home，在那里将能够使用数据库的节点。

## 15.2　Heroku 平台

Heroku 是使用平台即服务（PaaS）模型创建的最古老的云平台之一。与 Pivotal Cloud Foundry 相比，Heroku 没有内置的 Spring Cloud 应用程序支持，这会使开发人员的模型稍微复杂化，因为这意味着无法使用平台的服务来启用典型的微服务组件，包括服务发现、配置服务器或断路器。尽管如此，Heroku 仍包含了一些 Pivotal Web Services 无法提供的非常有趣的功能。

### 15.2.1　部署方法

开发人员可以使用命令行界面、Web 控制台或专用的 Maven 插件管理自己的应用程序。部署 Heroku 与部署 Pivotal 平台非常相似，但方法略有不同。主要方法假定开发人员通过从存储在本地 Git 存储库或 GitHub 上的源代码构建应用程序来部署应用程序。在将

分支中的某些更改推送到存储库之后，构建将由 Heroku 平台自动执行。或者，也可以根据所选分支中的最新版本代码的要求执行构建。部署应用程序的另一个有趣方法是将 Docker 镜像推送到 Heroku 的容器注册表。

### 1. 使用命令行界面

要使用命令行界面，首先需要下载 https://cli-assets.heroku.com/heroku-cli/channels/stable/heroku-cli-x64.exe（适用于 Windows）并安装 Heroku 命令行界面（CLI）。要使用命令行界面在 Heroku 上部署和运行应用程序，必须执行以下步骤。

（1）安装之后，可以使用 shell 中的命令 Heroku。首先，开发人员需要使用凭据登录 Heroku，如下所示。

```
$ heroku login
Enter your Heroku credentials:
Email: piotr.minkowski@play.pl
Password: ********
Logged in as piotr.minkowski@play.pl
```

（2）现在导航到应用程序的 root 目录并在 Heroku 上创建一个应用程序。运行以下命令后，不仅会创建应用程序，还会创建一个名为 heroku 的 Git 远程主机，这是与本地 Git 存储库相关联的，如下所示。

```
$ heroku create
Creating app... done, aqueous-retreat-66586
https://aqueous-retreat-66586.herokuapp.com/ |
https://git.heroku.com/aqueous-retreat-66586.git
Git remote heroku added
```

（3）此时可以通过将代码推送到 Heroku 的 Git 远程主机来部署应用程序。Heroku 将自动完成所有工作，如下所示。

```
$ git push heroku master
```

（4）如果应用程序成功启动，则开发人员将能够使用一些基本命令进行管理。按照如下所示的顺序，可以显示日志、更改正在运行的 Dynos 的数量（换句话说，其实就是扩展应用程序）、分配新的加载项（Addons），并列出所有已启用的加载项。

```
$ heroku logs --tail
$ heroku ps: scale web = 2
$ heroku addons: create mongolab
$ heroku addons
```

### 2. 连接到 GitHub 存储库

就个人而言，笔者更喜欢通过使用 GitHub 存储库连接到项目来将应用程序部署到 Heroku。这种部署方法同样有两种可能的方式：手动和自动。开发人员可以选择导航到应用程序详细信息面板上的 Deploy（部署）选项卡，然后将其连接到指定的 GitHub 存储库，如图 15.10 所示。如果单击 Deploy Branch（部署分支）按钮，则会立即从给定的 Git 开始构建并部署到 Heroku。或者，开发人员也可以通过单击 Enable Automatic Deploys（启用自动部署）按钮在所选分支上启用自动部署。此外，如果为 GitHub 存储库启用了 Heroku，则可以将 Heroku 配置为等待持续集成构建结果。这是一个非常有用的功能，因为它允许开发人员对项目运行自动化测试并确保它们在推送之前就已经通过。

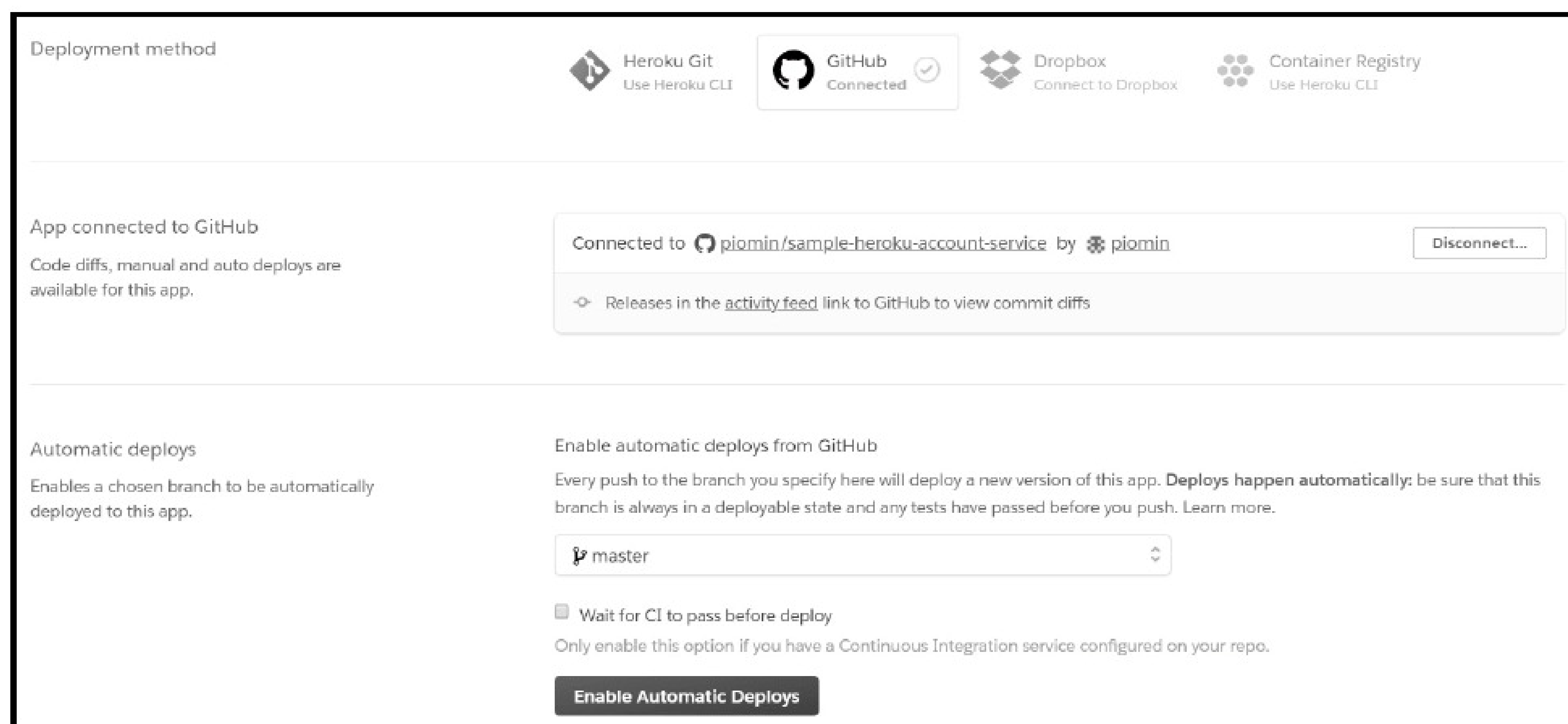



图 15.10　连接到 GitHub 存储库

### 3. Docker 容器注册表

Heroku 遵循了最新趋势，允许开发人员使用 Docker 部署容器化应用程序。为了能够做到这一点，开发人员应该在本地计算机上安装 Docker 和 Heroku 命令行界面。

（1）首先，需要通过运行 heroku login 命令登录 Heroku Cloud。下一步则是登录到 Container Registry（容器注册表）。

```
$ heroku container:login
```

（2）接下来，需要确保当前目录包含 Dockerfile。如果存在，则可以继续构建并通过执行以下命令将镜像推送到 Heroku 的 Container Registry。

```
$ heroku container:push web
```

（3）如果有现成的构建镜像，则可以标记它并将其推送到 Heroku。要完成此任务，需要通过执行以下命令来使用 Docker 的命令行（假设当前应用程序的名称是 piomin-order-service）。

```
$ docker tag piomin/order-service registry.heroku.app/piomin-order-
service/web
$ docker push registry.heroku.app/piomin-order-service/web
```

在成功推送镜像之后，新应用程序应在 Heroku 仪表板中可见。

## 15.2.2　准备应用程序

在将基于 Spring Cloud 组件的应用程序部署到 Heroku 时，开发人员不再需要对其源代码执行任何额外更改或添加任何其他库，因为这在按本地方式运行时就已经做了。这里唯一的区别在于配置设置，开发人员应该设置一个地址，以便将应用程序与服务发现、数据库或任何其他可以为微服务启用的加载项集成。当前示例与为 Pivotal 部署提供的示例相同，将数据存储在 MongoDB 中，该数据将作为 mLab 服务分配给应用程序。此外，在本示例中，每个客户端都会在 Eureka 服务器上注册它们自己，该 Eureka 服务器被部署为 piomin-discovery-service。我们的示例在 Heroku 上部署的应用程序列表如图 15.11 所示。

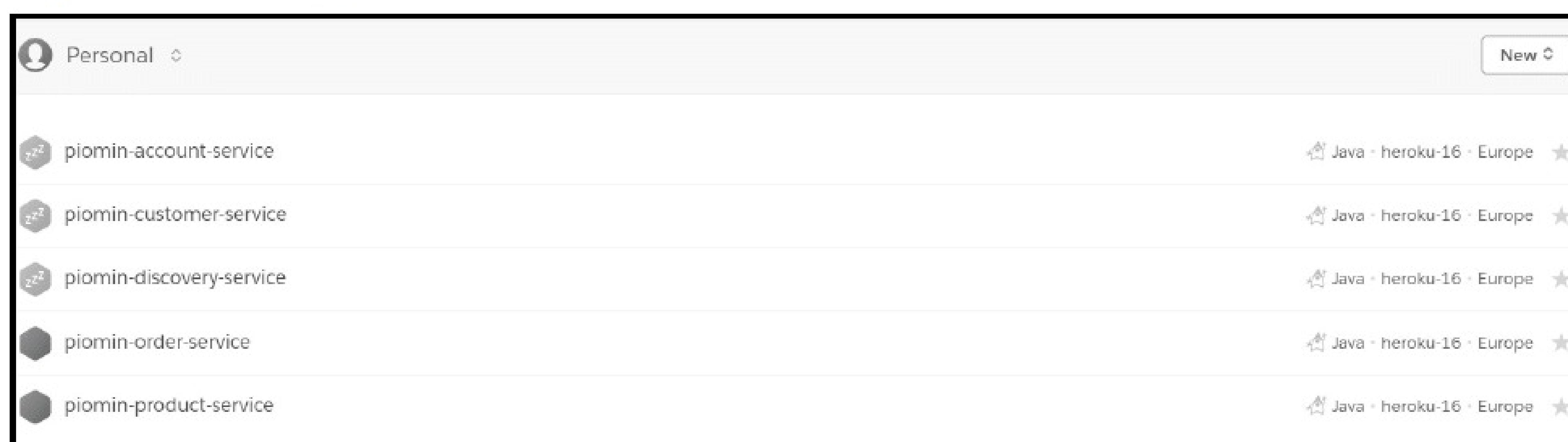



图 15.11　在 Heroku 上部署的应用程序列表

我们通过将应用程序与 GitHub 存储库连接的方式在 Heroku 上部署了这些应用程序。反过来，这需要为每个微服务创建一个单独的存储库。例如，order-service 服务的存储库可以在 https://github.com/piomin/sample-heroku-order-service.git 获得，其他微服务可能处于类似的地址下。开发人员可以轻松地分开这些微服务并将其部署在自己的 Heroku 账户上以执行测试。

现在以 account-service 应用程序为例，来仔细看一看如何提供配置设置。首先，我们必须使用 Heroku 平台提供的 MONGODB_URI 环境变量覆盖 MongoDB 的自动配置地址。

此外，还必须提供 Eureka 服务器的正确地址，以及在注册期间覆盖由发现客户端发送的主机名和端口。这是必需的操作，因为在默认情况下，每个应用程序都会尝试使用其他应用程序无法使用的内部地址进行注册。如果开发人员不覆盖这些值，则与 Feign 客户端的服务间通信将失败。

```
spring:
    application:
      name: account-service
    data:
      mongodb:
        uri: ${MONGODB URI}
  eureka:
    instance:
      hostname: ${HEROKU APP NAME}.herokuapp.com
      nonSecurePort: 80
    client:
      serviceUrl:
        defaultZone: http://piomin-discovery-service.herokuapp.com/eureka
```

请注意，环境变量 HEROKU_APP_NAME 是部署在 Heroku 上的当前应用程序的名称，如前面的代码段所示，且默认情况下不可用。要为应用程序启用变量（如 customer-service），可以使用实验性的加载项 runtime-dyno-metadata 运行以下命令。

```
$ heroku labs: enable runtime-dyno-metadata -a piomin-customer-service
```

## 15.2.3　测试部署

在部署完成之后，每个应用程序都可以在由其名称和平台域名组成的地址上使用，如 http://piomin-order-service.herokuapp.com。开发人员可以使用公开的 URL 地址（http://piomin-discovery-service.herokuapp.com/）调用 Eureka 仪表板，这将允许开发人员检查示例微服务是否已经注册。如果一切正常，则应该看到类似于图 15.12 所示的内容。

Instances currently registered with Eureka

| Application | AMIs | Availability Zones | Status |
|---|---|---|---|
| ACCOUNT-SERVICE | n/a (1) | (1) | UP (1) - 4786503c-2d95-43d3-a7f0-111186aa0692.prvt.dyno.rt.heroku.com:account-service:19265 |
| CUSTOMER-SERVICE | n/a (1) | (1) | UP (1) - bb2db4a3-2923-4fc4-a4ce-407c2b7d36be.prvt.dyno.rt.heroku.com:customer-service:35135 |
| ORDER-SERVICE | n/a (1) | (1) | UP (1) - 770d1584-9fe4-4536-8835-5d376f365a28.prvt.dyno.rt.heroku.com:order-service:43145 |
| PRODUCT-SERVICE | n/a (1) | (1) | UP (1) - d529d914-9da9-45e0-9b40-c4a0ce4b6364.prvt.dyno.rt.heroku.com:product-service:17541 |

图 15.12　已经注册的 Eureka 服务实例

由于每个微服务都已经公开了由 Swagger2 自动生成的 API 文档，因而可以访问/swagger-ui.html。打开 Swagger 用户界面仪表板（如 http://piomin-customer-service.herokuapp.com/swagger-ui.html），然后通过调用每个端点来轻松测试它们。customer-service 服务的 HTTP API 的可视化结果如图 15.13 所示。

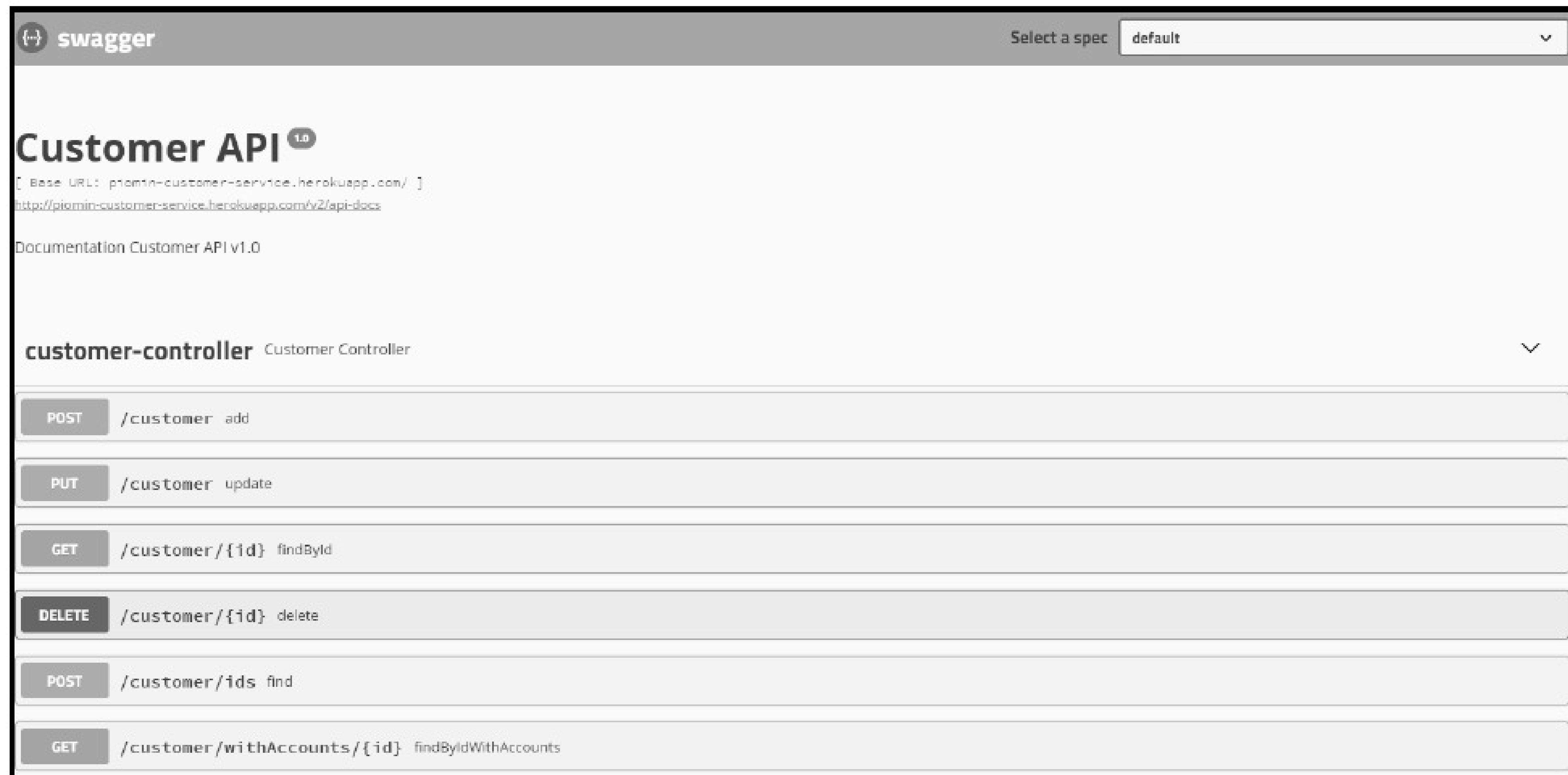



图 15.13　在 Swagger 用户界面仪表板查看服务的 API

每个微服务都会将数据存储在 MongoDB 中。可以通过添加 Heroku 提供的加载项（如 mLab）为项目启用此数据库。如前文所述，我们已经使用了相同服务的示例来在 Pivotal 平台上部署的应用程序中存储数据。要为应用程序启用加载项，可以找到每个应用程序的详细信息面板的 Resources（资源）选项卡，然后为选定的计划配置加载项。完成后，开发人员只需单击即可管理每个插件。对于 mLab 来说，将重定向到 mLab 站点（mlab.com），在该站点中可以看到所有集合、用户和生成的统计信息的列表。本示例的 mLab 仪表板如图 15.14 所示。

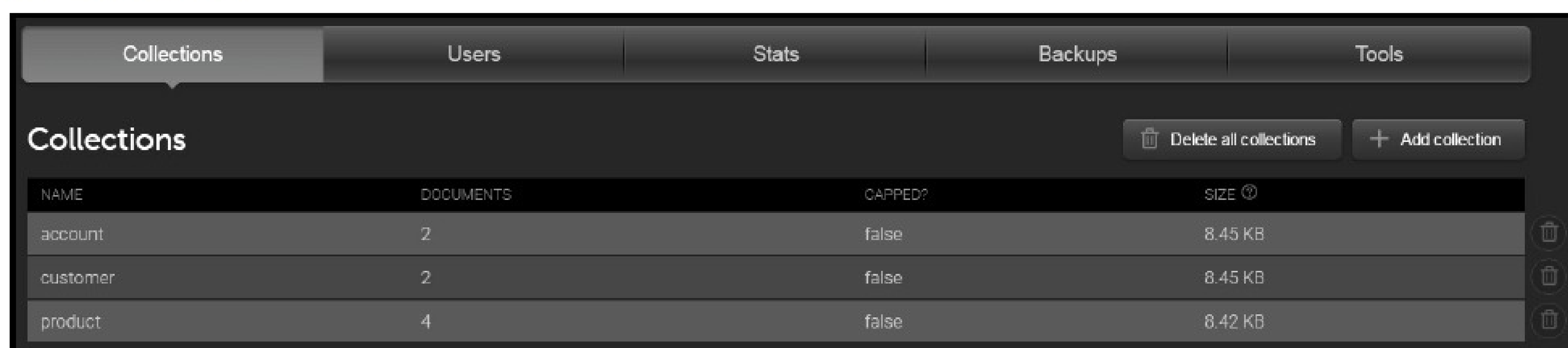



图 15.14　查看 mLab 仪表板

## 15.3　小　　结

我们已经到达 Spring Cloud 微服务之旅的终点！我们的练习始于本地计算机上的简单部署，但在第 14 章中，我们已经将微服务部署在由云供应商完全管理的环境中，它将自动构建、启动和公开指定域上的 HTTP API。笔者个人认为，使用任何最流行的编程语言或第三方工具（如数据库或消息代理），开发人员可以轻松地运行和扩展应用程序，以及公开应用程序之外的数据。事实上，我们每个人现在都可以在几个小时内实现并启动可用于生产模式的应用程序，而无须担心必须安装的软件。

本章展示了如何轻松地在不同平台上运行 Spring Cloud 微服务。通过示例演示了云原生应用程序的真正威力。无论是在笔记本电脑上以本地方式启动应用程序，还是在 Docker 容器内部或使用 Kubernetes，甚至是在 Heroku 或 Pivotal Web Services 等在线云平台上启动应用程序，开发人员都不必更改应用程序源代码中的任何内容，而仅需要在其属性中执行一些修改（假设在架构中使用了 Config Server，那么这些更改都不是侵入性的）。

在本书的最后两章中，我们研究了 IT 世界中的一些最新趋势。现在已经有越来越多的企业或组织开始讨论和使用诸如持续集成和持续交付（CI/CD）、使用 Docker 的容器化、使用 Kubernetes 的编排以及云平台之类的技术。实际上，这些解决方案也是微服务日益普及的部分原因。目前，该编程领域有一个领导者——Spring Cloud。Spring Cloud 所具有的功能之丰富，可以实现的与微服务相关的模式之多，目前尚无其他 Java 框架可以与之媲美。衷心希望本书能够帮助开发人员有效使用此框架，从而更好地构建和磨砺自己的基于微服务的企业系统。